

HOCHSCHULE RHEIN-WAAL
&
FLUXANA GMBH & CO. KG

BACHELOR'S THESIS

**Development of an automated powder
dosing system using a 6-DOF
collaborative robotic arm (cobot)**

by investigating the influence of vibration, angle of dosing,
and rotational speed to the mass flow of the powder.

Author:

Abdelrahman MOSTAFA
Matriculation No. 29528

Supervisor:

Prof. Dr. Ronny HARTANTO
Dr. Rainer SCHRAMM

*A thesis submitted in fulfillment of the requirements
for the Bachelor degree of Science*

in the

Mechatronic Systems Engineering
Faculty of Technology & Bionics

November 27, 2023

Declaration of Authorship

I, **Abdelrahman MOSTAFA**, declare that this thesis titled, "Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

HOCHSCHULE RHEIN-WAAL

Abstract

Faculty of Technology & Bionics
Research & Development at Fluxana GmbH & Co. KG

Bachelor of Science

**Development of an automated powder dosing system using a 6-DOF
collaborative robotic arm (cobot)**

by **Abdelrahman MOSTAFA**

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Declaration of Authorship	iii
Abstract	vii
Acknowledgements	ix
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	1
1.3 Thesis Structure	1
I Basics of Robotics, ROS, and Powder Dosage	3
2 Basics of Robotics	5
2.1 Types of Robots	6
2.2 Robot Components	8
2.3 Motion Planning	11
2.3.1 Motion Planning Framework (MPF)	11
2.3.2 URDF	12
URDF Components in Hydro	13
2.4 Forward & Inverse Kinematics	14
2.5 Robot Control (Software)	14
3 Robot Operating System ROS	15
3.1 Linux for Robotics	15
3.1.1 What Is Ubuntu? and Why for Robotics?	15
3.1.2 Ubuntu File Structure	16
3.2 Philosophy Behind ROS	17
3.3 Preliminaries	18
3.3.1 ROS-Graph [6]	18
3.3.2 Roscore [58]	20
3.3.3 catkin, Workspaces, and ROS Packages	21
3.4 Nodes in ROS	23
3.4.1 Benefits of Using Nodes	23
3.4.2 Node Identification	23
3.4.3 Node Type	23
3.5 ROS Master Communication	24
3.5.1 .bashrc File	25
3.6 Publishers-Subscribers	25
3.6.1 Topics in ROS	26
Key Characteristics:	26
3.6.2 Publishing a topic in ROS node	27

3.6.3	Messages in ROS	28
3.6.4	Subscribing to a topic in ROS node	29
3.7	Services	30
3.7.1	Service-Clients node in ROS	30
3.7.2	*.srv Files	31
3.7.3	Configuration for Custom Service Compilation Modification of CMakeLists.txt	31
	Modification of package.xml	31
	Modification of package.xml	33
3.7.4	Service-Server node in ROS	34
3.8	Actions	37
3.9	ROS commands summary	38
3.10	MoveIt! [47]	39
4	Basics of Powder Dosing	41
4.1	Affecting Parameters	41
II	Experimental Set-up, Methodology, and Results	43
5	Methodology	45
5.1	Sequence Logic	45
5.2	Important Functions from FX_ROS Package	46
5.3	State Diagram	47
5.3.1	Zero State	47
5.3.2	Calibration	48
5.3.3	Retrieve the Pre-Saved Trajectories	49
5.3.4	Balance Set-Up	51
5.3.5	Obtain the Crucible	52
5.3.6	Do the Experiments	52
5.3.7	Return the Crucible	52
5.3.8	Unset the Balance	52
5.3.9	Go home	52
6	Experimental Set-up	53
6.1	Frame	53
6.2	Crucibles	55
6.3	Balance Device	56
6.4	Ned2 Collaborative Robot	56
6.4.1	Hardware Configurations	56
	Cobot's Gripper	59
6.4.2	Software Tools	59
	ROS-Melodic	60
6.5	Precision Validation	61
6.6	Vibration Motor	61
7	Evaluation & Results	63
7.1	Evaluation	63
8	Conclusion & Future Work	65
8.1	Future Work	65

A Important Python Fuctions Used in this Project	67
A.1 Calling a Service Server	67
A.1.1 Example of Call_Aservice Function	68
A.2 Subscribing to Topic	68
A.2.1 Examples of Subscribe Function	68
A.3 Moveing the Cobot using Joints	69
A.4 Moveing the Cobot using Pose	70
A.5 Use Forward Kinematics	72
B FX_ROS.py Library in Python	73
C xlsx.py Package in Python	81
D Balance Technical Drawing	85
E Glass and Metal Crucibles Technical Drawing	89
F Frame Technical Drawing	93
Bibliography	99

List of Figures

2.1 Examples of a humanoid and a collaborative robot.	6
2.2 Manipulators with 3 DoF of movement.	8
2.3 Illustration of different joints. [20]	8
2.4 The Wrist and End-effector Parts. [20]	9
2.5 How a Cobot range of motion and DoF are inspired by human motion. [25]	10
2.6 High level diagram of various planning components	11
2.7 URDF diagram	13
3.1 Ubuntu file system structure	16
3.2 Graphical representation of a ROS system	19
3.3 Roscore Connections with the nodes	20
3.4 Files' structure for ROS workspace. [2]	21
3.5 Master Name Service Example [59]	24
3.6 Publisher–Subscriber communication model.	26
3.7 Server–Client Service communication	30
3.8 Server–Client Action communication	37
5.1 The sequence logic describing the full dosing process	47
5.2 Calibration process of Ned2 cobot.	48
5.3 The angle α with reference to the $x - axis$.	50
5.4 Defining the pre-named positions used in the trajectory planning.	50
5.5 Indexing the crucibles' holders with numerical values.	50
5.6 Balance Setup flowchart	51
6.1 The Full Framework of System	53
6.2 Demonstration of the parts of the framework.	54
6.3 The glass crucible with its hangers.	55
6.4 The metal crucible and its hanger.	55
6.5 The balance with its metal plates and rubber buffers.	56
6.6 Ned2 cobot provided by Niryo [14].	56
6.8 The detailed workspace of Ned2 cobot.	57
6.7 The Servo motors used in Ned2.	57
6.9 The development phase of the gripper.	59
6.10 The main tasks of the gripper.	59
6.11 Software tools supported by Niryo in Ned2. [49]	60
6.12 The Catkin workspace in Ned2	60

List of Tables

2.1	Robots Classification	7
3.1	ROS commands summary with some examples context.	39
5.1	Most important functions used in the research	46
5.2	The positions used to plan the trajectories.	49
6.1	Technical Specification of Ned2 [13]	58

List of Abbreviations

API	Application Programming Interface
BASH	Bourn Again SHell
Cobot	Collaborative Robot
DoF	Degree of Freedom
FK	Forward Kinematics
GUI	Grafical User Interface
IK	Inverse Kinematics
KDL	Kinematics and Dynamics Library
MPF	Motion Planning Framework
OpenCV	Open Source Computer Vision Library
ROS	Robot Operating System
TCP	Tool Center Point
TCP/IP	Transmission Control Protocol/Internet Protocol
SSH	Secure Shell
URDF	Unified Robot Description Format
TF	TransFormation
URDF	Unified Robot Description Format

Chapter 1

Introduction

1.1 Motivation

1.2 Objectives

1.3 Thesis Structure

Part I

Basics of Robotics, ROS, and Powder Dosage

Chapter 2

Basics of Robotics

As an academic field, robotics emerges as a relatively youthful discipline, characterized by profoundly ambitious objectives, the most paramount of which is the creation of machines capable of emulating human behavior and cognitive processes. This quest to engineer intelligent machines inherently compels us to embark on a journey of self-exploration. It prompts us to scrutinize the intricacies of our own design—why our bodies possess the configurations they do, how our limbs synchronize in movement, and the mechanisms behind our acquisition and execution of intricate tasks. The realization that the fundamental inquiries in robotics are intrinsically linked to inquiries about our own existence forms a captivating and immersive aspect of the robotics pursuit. [29]

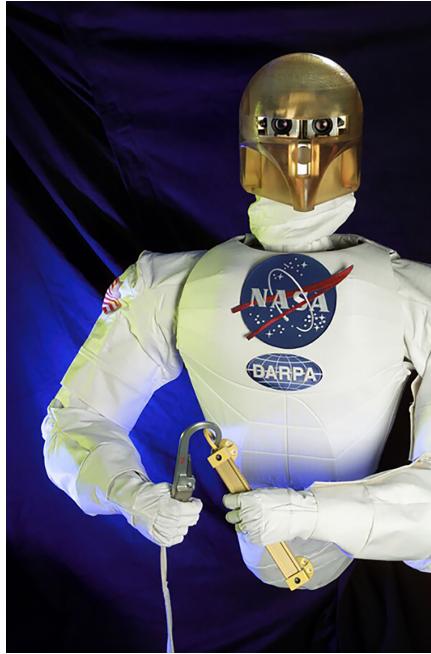
Definition ➤

Robotics is the scientific field dedicated to the study of robots—machines capable of autonomous operation, carrying out various tasks without direct human intervention. [28]

While science fiction often envisions robots in humanoid or android forms, real-world robots, especially those designed for industrial applications, typically deviate from human physical resemblance. These robots typically comprise three fundamental components: a mechanical structure, often represented by a robotic arm, enabling physical interaction with the robot's environment or itself; sensors that collect data on various physical attributes such as sound, temperature, motion, and pressure; and a processing system that interprets data from the robot's sensors, providing instructions for task execution.

It's worth noting that certain devices, like web-crawling search engine bots that systematically explore the internet to collect information on links and online content, may lack physical mechanical elements. Nonetheless, they are still classified as robots because they exhibit the ability to perform repetitive tasks autonomously.

This chapter delves into an exploration of various robot classifications, delving into the foundational principles of mechanics and kinematics. It also scrutinizes the intricacies of planning and control within the context of collaborative robots (cobots). The knowledge presented in this chapter draws significant inspiration from two primary sources, namely '**Modern Robotics**' [29] by Kevin M. Lynch and Frank C. Park, and '**Theory of Applied Robotics**' [20] authored by Professor Reza N. Jazar. For a deeper understanding of these topics, I encourage you to refer to these texts.



(a) NASA Robonaut [1] [4]



(b) Universal Robot (UR20) [54]

FIGURE 2.1: Figure (a) illustrates an instance of a humanoid robot developed by NASA, while Figure (b) exemplifies a cobot.

2.1 Types of Robots

Across diverse industries, robotics solutions have emerged as catalysts for heightened productivity, elevated safety standards, and increased operational adaptability. Organizations at the vanguard of innovation are discerning forward-looking applications of robotics that yield palpable and quantifiable outcomes. Intel collaborates closely with manufacturers, system integrators, and end-users, actively contributing to the realization of robots that deliver impactful, human-centered results.

According to an article from Intel regarding the classification of robots [18], the current generation of robots has been categorized into six distinct groups.

Autonomous Mobile Robots (AMRs) [10] AMRs navigate their environments and make rapid decisions on the fly. These robots employ advanced technologies like sensors and cameras to gather data from their surroundings. Equipped with onboard processing capabilities, they analyze this data and make well-informed decisions—whether it involves avoiding an approaching human worker, selecting the exact parcel to pick, or determining the suitable surface for disinfection. These robots are self-sufficient mobile solutions that operate with minimal human intervention. [43]

Automated Guided Vehicles (AGVs) [57] While AMRs navigate their surroundings autonomously, AGVs typically operate along fixed tracks or predetermined paths and frequently necessitate human supervision. AGVs find extensive application in scenarios involving the transportation of materials and goods within controlled settings like warehouses and manufacturing facilities.

Humanoids [26] While numerous mobile humanoid robots could, in a technical sense, be classified as Autonomous Mobile Robots (AMRs), this categorization

primarily applies to robots fulfilling human-centric roles, frequently adopting human-like appearances. These robots leverage a similar array of technological components as AMRs to perceive, strategize, and execute tasks, encompassing activities such as offering navigational assistance or providing concierge services.

Hybrids [46] Diverse categories of robots are frequently integrated to engineer hybrid solutions that possess the capacity to execute intricate operations. For instance, the fusion of an AMR with a robotic arm can yield a versatile system tailored for the handling of packages within a warehouse environment. As functionalities are amalgamated within single solutions, there is a concurrent consolidation of computational capabilities.

Articulated Robots [45] Commonly referred to as robotic arms, are designed to replicate the versatile functions of the human arm. These systems typically incorporate a range of rotary joints, varying from two to as many as ten. The inclusion of additional joints or axes equips these robotic arms with a wider range of motion capabilities, rendering them particularly well-suited for tasks such as arc welding, material manipulation, machine operation, and packaging.

Cobots [37] Collaborative Robots, commonly referred to as cobots, are engineered with the specific purpose of working in tandem with, or directly alongside, human operators. Unlike many other categories of robots that function autonomously or within strictly segregated workspaces, cobots share work environments with human personnel to enhance their collective productivity. Their primary role often involves the removal of manual, hazardous, or physically demanding tasks from daily operations. In certain scenarios, cobots are capable of responding to and learning from human movements, further enhancing their adaptability.

The initial four robots fall under the category of mobile robots, possessing the capability to navigate within their surroundings, while the latter two are categorized as stationary robots, as detailed in table 2.1 below.

TABLE 2.1: Robots Classification.

Mobile	Stationary
AMRs	
AGVs	Articulated robots
Humanoids	Cobots
Hybrids	

Within the scope of this paper, our exclusive focus will be on **cobots** [37]. Across all the experiments conducted in this study, a cobot (Ned2, detailed and described in chapter 6) has been consistently utilized.

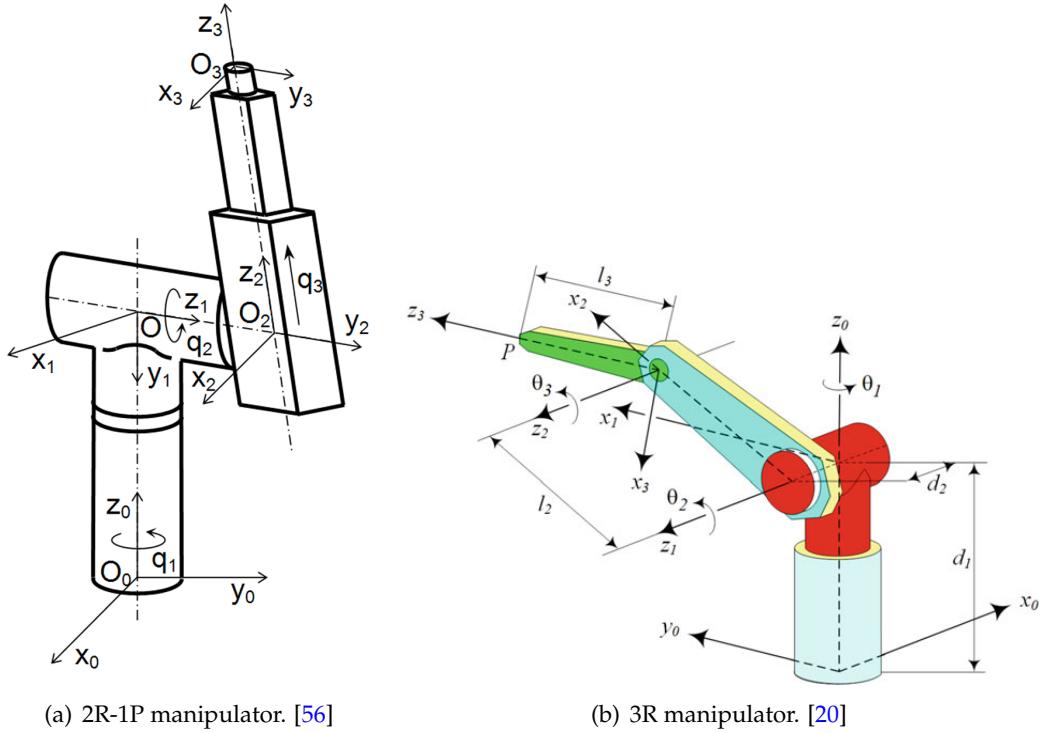


FIGURE 2.2: Manipulator with 3 DoF of movement.

2.2 Robot Components

In our study, we establish a kinematic model for a robotic manipulator, which is essentially a multi-body system comprising interconnected rigid bodies. These bodies are connected through revolute or prismatic joints (only revolute in our application), enabling relative movement. We employ principles of rigid body kinematics to elucidate the relative motions between these interconnected bodies.

It's imperative to note that a comprehensive robotic system encompasses not only the manipulator or rover but also components such as the wrist, end-effector, actuators, sensors, controllers, processors, and software. [20]

Link: In the realm of robotics, each individual rigid component within a robot that

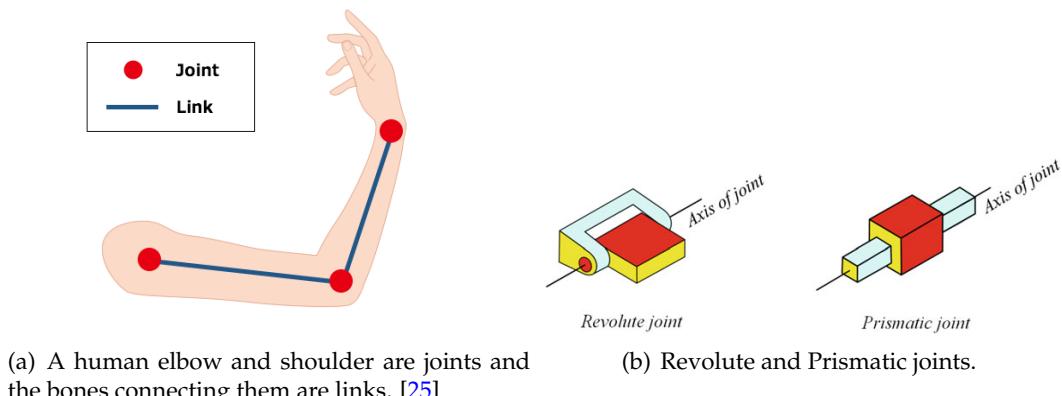


FIGURE 2.3: Illustration of different joints.

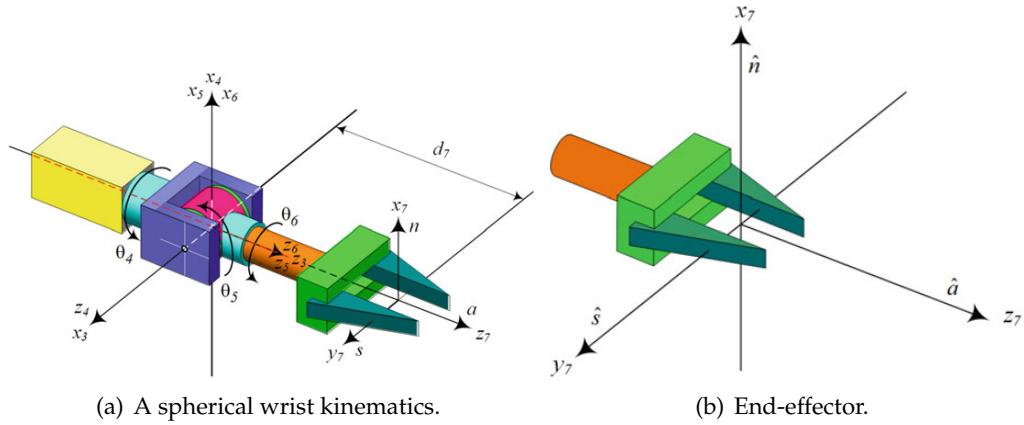


FIGURE 2.4: The Wrist and End-effector. [20]

possesses the capacity to move concerning all other components is formally known as a 'link.' This terminology accommodates various descriptions, including 'bar,' 'arm,' or any object deemed equivalent to a link in the context of robot mechanics. A robot arm or link, in essence, represents a solid, rigid element capable of relative motion when compared to the other links within the robotic structure.

Moreover, when we encounter two or more linked components that are entirely constrained in terms of relative movement, they are collectively regarded as a 'compound link,' forming a unified and motionally inseparable entity within the robot's framework.

Joint: A kinematic pair refers to two rigid bodies that are in constant contact with one another, enabling potential relative motion. These pairs connect two links at a joint where their relative movement is described by a single joint coordinate. Typically, joints manifest as revolute (rotary) or prismatic (translatory). In Figure 2.3(a), the illustration portrays joints and links within a human arm. A revolute joint (R) mimics a hinge, enabling relative rotation between two links, while a prismatic joint (P) facilitates translational relative motion between the two links.

The relative rotation of interconnected links through a revolute joint transpires along an axis referred to as the joint axis. Similarly, the translation between two connected links via a prismatic joint occurs along an axis, also known as the joint axis. The joint coordinate, or joint variable, represents the value describing the relative position of these connected links at the joint. For a revolute joint, it represents an angle, while for a prismatic joint, it denotes a distance. The depiction of revolute and prismatic joints is respectively shown in Figures 2.3(b).

Manipulator: The primary structure of a robot comprises links, joints, and associated structural components, known as the manipulator. A manipulator is transformed into a robot when integrated with a wrist, gripper, and its control system. However, in the literature, robots and manipulators are often used interchangeably, both denoting robotic systems. In Figure 2.2(b), a 3R manipulator is depicted, while Figure 2.2(a) showcases a 2R-1P manipulator arm.

Wrist: The joints situated in the kinematic chain between the forearm and the end-effector are termed the wrist. It's customary to design manipulators with

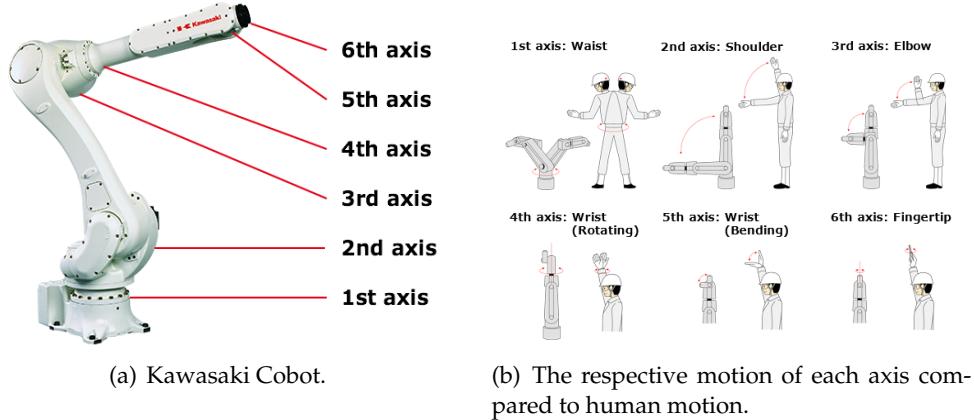


FIGURE 2.5: How a Cobot range of motion and DoF are inspired by human motion. [25]

spherical wrists, comprising three intersecting revolute joint axes that converge at a common point known as the wrist point. Illustrated in Figure 2.4(a) is a stationary spherical wrist composed of three revolute joints with orthogonal rotation axes. The spherical wrist significantly streamlines kinematic analysis by separating the end-effector's positioning and orientation. Consequently, the manipulator's wrist point holds three degrees of freedom for positioning, controlled by three arm joints. The orientation's degrees of freedom depend on the wrist design, which may involve one, two, or three degrees based on the specific application.

End-Effector: The end-effector constitutes the segment attached to the final link that carries out the designated tasks of the robot. Typically, the simplest end-effector is a gripper, allowing two actions: opening and closing. Both the arm and wrist assemblies primarily facilitate the positioning of the end-effector and any accompanying tools. It is the end-effector that executes the intended tasks. Considerable research is dedicated to developing specialized end-effectors and tools, including anthropomorphic hands designed for prosthetic purposes in manufacturing. Consequently, a robot is comprised of a manipulator or mainframe, a wrist, and an end-effector, while the wrist and end-effector assembly is sometimes referred to as a hand. Figure 2.4 presents an example of an end-effector.

Actuator: Actuators serve as the driving force akin to a robot's muscles, enabling the alteration of their configuration. They supply the necessary power to act on the mechanical structure, counteracting external forces like gravity and inertia, to adjust the geometric position and orientation of the robot's hand. Actuators, typically electric, hydraulic, or pneumatic, must be controllable.

Sensors: Sensors are components employed to observe and gather data related to both internal and external conditions. Within the context of this book, pivotal data such as joint positions, velocities, accelerations, and forces hold significant importance for measurement. Embedded within the robot system, these sensors relay pertinent details concerning each link and joint to the control unit. The control unit, based on this information, ascertains the robot's configuration.

Controller: The robot's controller, or control unit, fulfills three primary functions:

1. Information role, involved in the collection and processing of data obtained from the robot's sensors.
2. Decision role, responsible for planning the geometric movement of the robot structure.
3. Communication role, tasked with managing information exchange between the robot and its environment. The control unit encompasses both the processor and software components.

2.3 Motion Planning

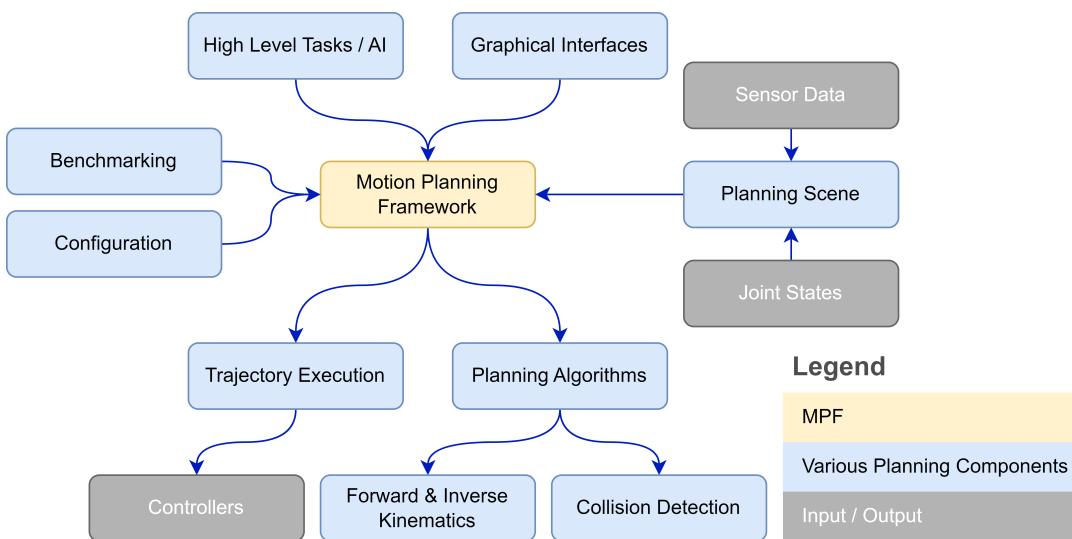


FIGURE 2.6: High level diagram of various planning components in a Motion Planning Framework (MPF). [7]

The realm of robotic motion planning has emerged as an integral and evolving discipline in the field of robotics [31]. Its focus is on the systematic transformation of task directives into isolated motion sequences that allow machines to navigate environments confining them.

The software development of a motion planning framework (MPF) is challenging and involves combining many disparate fields of robotics and software engineering [36]. In this context, the software is labeled as a framework because of its ability to generalize the components of motion planning through interfaces, which we will further examine later on.

2.3.1 Motion Planning Framework (MPF)

A variety of software components, known as planning components, are necessary for an MPF to function along with common data structures. The planning components are illustrated in Figure 2.6, and involve numerous interactions. Among them is a planning component that carries out motion planning. It encompasses one or several algorithms suitable for resolving the challenges met by a robot. Although the scope of motion planning is vast, there is no unified formula catering to all situations. As such, a framework free of robot constraints should comprise different algorithms as well as their variants.

Several important aspects must be considered in robot planning. First, a collision-checking module is necessary to identify possible intersections of geometric objects in the planning environment and the robot model. Additionally, a forward kinematics solver is essential for establishing the robot's geometry based on joint positions, while an inverse kinematics solver becomes necessary when some planning approaches rely on Cartesian end effector space. Different constraints, such as joint/velocity/torque limits and stability requirements, may also require other components.

To create a truly effective MPF, the inclusion of supplementary elements is essential. Upon reaching a configuration space solution, articulated through position waypoints, a corresponding timeframe-based trajectory max needs to be integrated for proper execution. At runtime, a controller manager selects the appropriate joint-related controllers from its available resources for any given path parameter of a trajectory. Alongside the recognition data gathered through its priority perception pipeline, integrated secondary components in MPF-assembled object ownership need to augment object perception awareness and provide additional diagnostics data. Meanwhile, application-specific taskings like pick-and-place ultimately yield the above by taking into account the respective contributions of said motion-planning string instruments. Additional encapsulated MOF behaviors available as add-ons include debugging and analysis tools, widespread testing suites, as well as the temptation of the discernible user screen perception. [7]

2.3.2 URDF

Definition

Robot Model Format: The robot model in robotics serves as a structured representation encompassing a robot's three-dimensional design, kinematics, and various properties, such as geometric visualization meshes, collision geometry for efficient collision checking, joint limits, sensors, and dynamic characteristics like mass, moments of inertia, and velocity limits.

While the typical representation involves a kinematic tree illustrating relationships between joints and links, this approach encounters challenges with robots featuring closed chains. For the purposes of our application and many modern Motion Planning Frameworks (MPFs), we define modeled robots as arbitrarily articulated rigid bodies.

Effective robotics software demands a standardized format capable of expressing diverse hardware configurations while being user-friendly for setup. In MoveIt! (a ROS motion planning framework, explained in section 3.10) for example, this requirement is met by adopting the **Unified Robotic Description Format (URDF [35])** Document Object Model. This data structure is populated through the interpretation of human-readable XML schemas, including URDF-formatted files (distinct from the data structure) and the industry-standard Collada [19] format.

Creating a precise robot model can be a challenging task. While URDF models for many robots are readily available, facilitating user avoidance of this challenge, custom robots necessitate the development of a new robot model. In such cases, the URDF model in ROS proves to be the most suitable choice, providing users with tools for XML validation, visualization, and the direct conversion of a SolidWorks CAD model into URDF format [3].

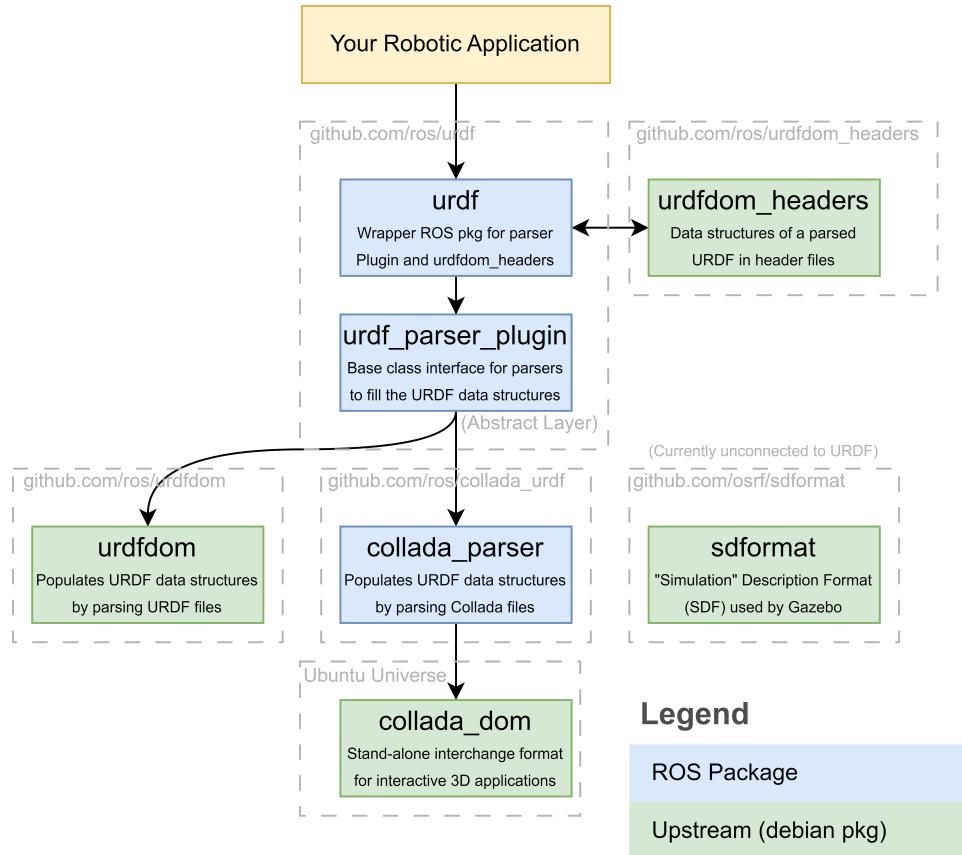


FIGURE 2.7: A diagram that attempts to explain how a number of different packages and components make up URDF [35]

URDF Components in Hydro

- The core URDF parser and data structures (`urdfdom`, `urdfdom_headers`) are now stand-alone packages.
- These packages are separated from ROS (check next chapter 3 for more details) dependencies and are slated for independent release into Ubuntu.
- A clear distinction exists between a URDF file and a URDF data structure:
 - A URDF file adheres to the XML format outlined on the <http://wiki.ros.org/urdf/XML>.
 - A URDF data structure comprises a set of versatile classes for parsing various formats (currently URDF and Collada).
- The introduction of a new plugin abstraction layer (`urdf_parser_plugin`) facilitates populating URDF data structures with different file formats, presently supporting URDF and Collada.
- ROS packages `urdf_parser` and `urdf_interface` have been deprecated in Groovy and removed in Hydro, reflecting ongoing optimizations in URDF handling within ROS.

2.4 Forward & Inverse Kinematics

As shown in Figure 2.6, an important part of the motion planning algorithms are forward and inverse kinematics.

2.5 Robot Control (Software)

Chapter 3

Robot Operating System | ROS

Definition

ROS serves as an open-source, meta-operating system designed to cater to the requirements of your robot. Much like traditional operating systems, it offers essential functionalities such as hardware abstraction, precise control over low-level devices, integration of commonly-used features, seamless message exchange between processes, and streamlined package management. Furthermore, ROS equips you with a comprehensive set of tools and libraries to facilitate tasks like code acquisition, compilation, development, and execution across diverse computing environments. [41]

In this chapter, we will delve into the foundational elements of the ROS framework. We will begin by exploring the relevance and basics of Linux and Ubuntu in the context of ROS. Following that, we will delve into the philosophical underpinnings of ROS, its master communication structure, and how these elements relate to topics, services, and actions. Finally, we will provide an explanation of the motion planning framework known as MoveIt.

All the code samples used in the chapter to explain ROS main concepts are either written by me, taken from the referenced books and articles, or cited from [the Construct](#) platform for robotics.

3.1 Linux for Robotics

Linux is a free, open-source operating system that includes several utilities that will significantly simplify your life as a robot programmer. As will be shown in the upcoming sections, ROS (Robot Operating System) is based on a Linux system. All commands and concepts explained here are taken from the Linux tutorial made by the University of Surrey. [23]

3.1.1 What Is Ubuntu? and Why for Robotics?

Ubuntu, accessible at www.ubuntu.com, stands as a widely acclaimed Linux distribution rooted in the Debian architecture (source: <https://en.wikipedia.org/wiki/Debian>). Notably, it's freely available and open source, permitting extensive customization for specific applications. Ubuntu boasts an extensive software repository, comprising over 1,000 software components, encompassing essentials such as the Linux kernel, GNOME/KDE desktop environments, and a suite of standard desktop applications, including word processing tools, web browsers, spreadsheets, web servers, programming languages, integrated development environments (IDEs), and

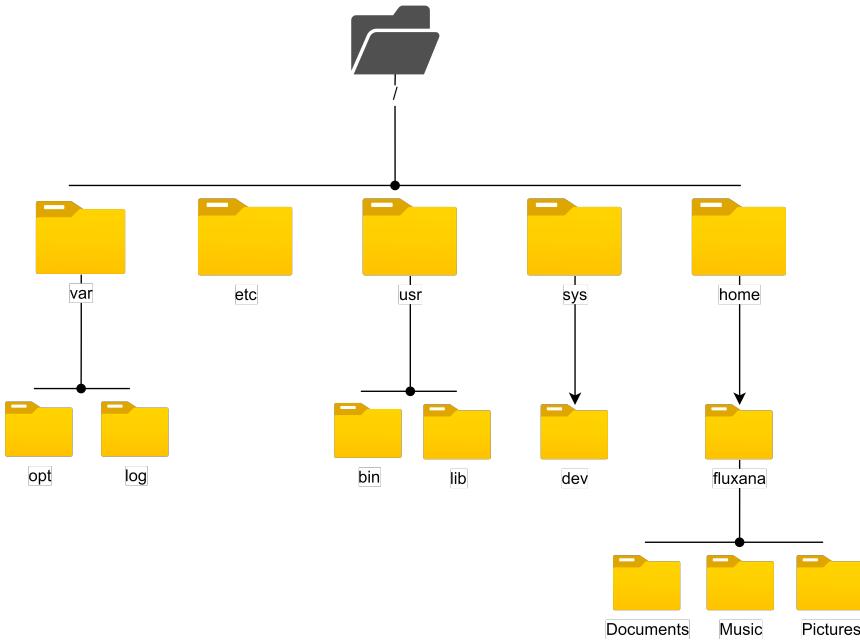


FIGURE 3.1: Ubuntu file system structure

even PC games. Versatile in its deployment, Ubuntu can operate on both desktop and server platforms, accommodating architectures like Intel x86, AMD-64, ARMv7, and ARMv8 (ARM64). Canonical Ltd., headquartered in the UK (www.canonical.com), provides substantial backing to Ubuntu. [27]

In the realm of robotics, software stands as the nucleus of any robotic system. An operating system serves as the foundation, facilitating seamless interaction with robot actuators and sensors. A Linux-based operating system, such as Ubuntu, offers unparalleled flexibility in interfacing with low-level hardware while affording provisions for tailored OS configurations tailored to specific robot applications. Ubuntu's merits in this context are manifold: it exhibits responsiveness, maintains a lightweight profile, and upholds stringent security measures. Additionally, Ubuntu boasts a robust community support ecosystem and a cadence of frequent releases, ensuring its perpetual relevance. It also offers long-term support (LTS) releases, guaranteeing user assistance for up to five years. These compelling attributes have cemented Ubuntu as the preferred choice among developers in the Robot Operating System (ROS) community. Indeed, Ubuntu stands as the sole operating system that enjoys comprehensive support from ROS developers. The Ubuntu-ROS synergy emerges as the quintessential choice for programming robots. [23]

3.1.2 Ubuntu File Structure

Similar to the 'C drive' in a Windows operating system, Linux incorporates a dedicated storage area for its system files, known as the root file system. This root file system is established during the Ubuntu installation process, with the assignment of '/' as its designated mount point. For a visual representation of the Ubuntu file system architecture, refer to Figure 3.1.

The following describes the uses of each folder in the file system:

- */bin and /sbin*: These directories house essential system applications, akin to the 'C:/Windows' folder in Windows.

- */etc*: Within this directory, system configuration files are stored.
- */home/yourusername*: Equivalent to the 'C:/Users' directory in Windows, this directory serves as the user's home.
- */lib*: Similar to '.dll' files in Windows, the '/lib' directory contains library files.
- */media*: This directory serves as the mount point for removable media.
- */root*: The '/root' directory contains files associated with the root user, who holds administrative privileges in the Linux system.
- */usr*: Pronounced 'user,' the '/usr' directory hosts a majority of program files, akin to 'C:/Program Files' in Microsoft Windows.
- */var/log*: Within this directory, you'll find log files generated by various applications.
- */home/yourusername/Desktop*: The location for Ubuntu desktop files.
- */mnt*: Mounted partitions are accessible in this directory.
- */boot*: This directory stores essential files required for the boot process.
- */dev*: Linux device files are located here.
- */opt*: The '/opt' directory serves as the designated location for optionally installed programs. (For instance, ROS is installed in '/opt').
- */sys*: This directory houses files containing critical information about the system.

3.2 Philosophy Behind ROS

The philosophical objectives of ROS can be succinctly described as follows [39]:

- Decentralized collaboration: Emphasizing **peer-to-peer** interactions. ROS consists of multiple interconnected software components enabling continuous message exchange without centralized routing, allowing for scalability even with increased data volume, although it may add complexity to the system infrastructure.
- **Tool-oriented** approach: Focusing on the development of a robust set of tools. ROS, influenced by Unix's architectural principles, constructs complex software systems through multiple small, versatile programs, differentiating itself from other robotics software by decentralizing various tasks to discrete software, enabling continual evolution and tool enhancement for specific task domains.
- **Multilingual support**: Enabling compatibility with multiple programming languages. ROS embraces a multilingual approach, allowing modules to be written in diverse programming languages, including C++, Python, LISP, Java, JavaScript, MATLAB [8], Ruby, Haskell, R, Julia, and others, with this book predominantly using the Python client library for code examples, yet noting the flexibility to utilize other available client libraries for discussed tasks.

- **Thin design:** Prioritizing a streamlined framework. ROS conventions encourage an approach where developers build standalone libraries that can be incorporated into ROS modules, fostering versatile software reuse and enabling streamlined automated testing via established continuous integration tools.
- **Openness and freedom:** Being freely available and based on **open-source** principles. ROS, operating under the permissive BSD license [40], enables diverse licensing arrangements by allowing flexible integration of closed-source and open-source modules, accommodating commercial, academic, and hobby projects, ensuring compliance within varied environments [39].

To the best of our knowledge, no existing framework encompasses this specific set of design principles. This section aims to delve into these philosophies, elucidating how they have profoundly influenced the design and implementation of ROS [39].

3.3 Preliminaries

Before delving into ROS, it's essential to introduce the fundamental concepts that underpin this framework. ROS systems are constituted by a multitude of autonomous programs that maintain continuous communication with one another. This section provides an in-depth exploration of this architectural setup and the associated command-line tools. It further delves into the intricate aspects of ROS naming conventions and namespaces, demonstrating their role in facilitating code reusability. [38]

3.3.1 ROS-Graph [6]

One of the original challenges inspiring the creation of ROS was commonly known as the 'fetch an item' problem. This scenario involved a relatively large and complex robot equipped with various sensors, a manipulator arm, and a mobile base. In the 'fetch an item' problem, the robot's objective is to navigate a typical home or office environment, locate a specified item, and transport it to the designated location. This task led to several key observations, which subsequently became foundational design goals for ROS:

- The application task can be broken down into numerous autonomous subsystems, encompassing areas like navigation, computer vision, and grasping.
- These subsystems are adaptable for various tasks, such as security patrols, cleaning, and mail delivery, among others.
- By implementing appropriate hardware and geometry abstraction layers, the majority of application software can be made compatible with different robotic platforms.

These principles are exemplified through the core structure of a ROS system: its graphical representation. In ROS, multiple programs operate concurrently and communicate by exchanging messages. This system structure is conveniently portrayed as a mathematical graph, with nodes representing individual programs and edges indicating their communication. While Figure 3.2 illustrates a sample ROS graph from one of the early 'fetch an item' implementations, the specific details are less significant compared to the overarching concept of a ROS system as an assembly of

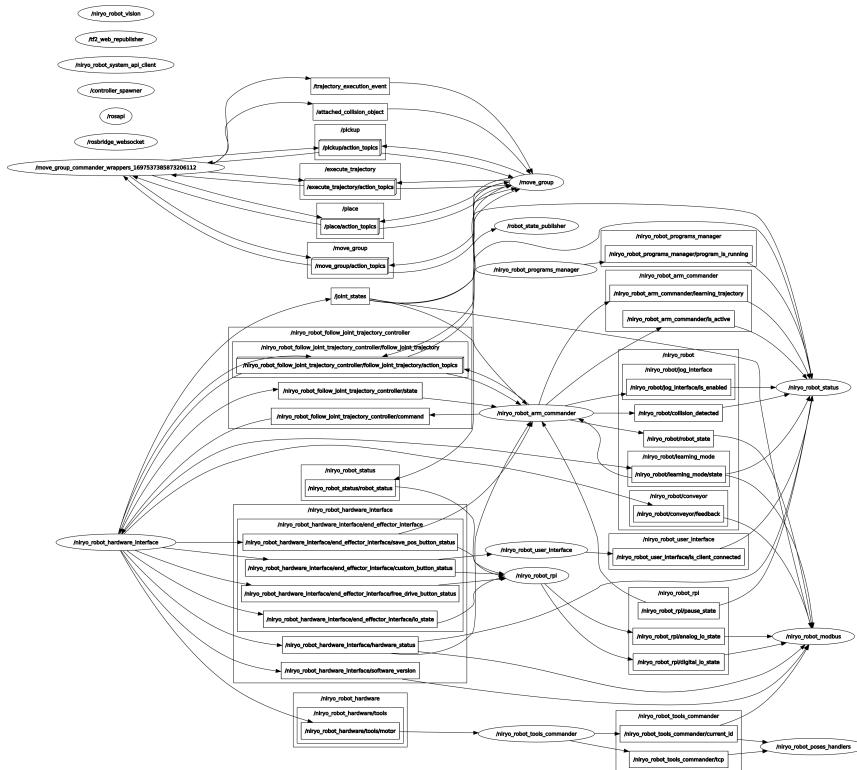


FIGURE 3.2: Graphical representation of a ROS system for ‘Niryo-Ned2 [13]’ robot—nodes/topics within the graph symbolize individual software modules, while edges denote message streams facilitating the exchange of sensor data, actuator commands, planner states, intermediate representations, and other relevant information.

nodes engaged in message-based communication. This representation serves as a practical framework for software development, emphasizing the modular nature of ROS programs, or 'nodes,' as integral components within a larger system.

In summary, within a ROS graph, a node signifies a software module engaged in message transmission, and an edge denotes the flow of messages between two nodes. While complexity can increase, nodes are typically POSIX processes, and edges are akin to TCP connections, enhancing fault tolerance as a software crash typically affects only the crashing process, leaving the rest of the graph operational. The events leading to the crash can often be reconstructed by logging messages entering a node and replaying them within a debugger at a later time.

One of the most significant advantages of a loosely coupled, graph-based architecture is the capacity to rapidly prototype complex systems with minimal or no need for additional 'glue' software during experimentation. Individual nodes, such as the object recognition node in a 'fetch an item' system, can be effortlessly replaced by launching an entirely different process that handles images and generates labeled objects. Beyond node replacement, entire segments of the graph (subgraphs) can be dynamically dismantled and substituted with other subgraphs in real time. This flexibility extends to replacing real-robot hardware drivers with simulators, swapping navigation subsystems, fine-tuning algorithms, and more. Since ROS dynamically generates the necessary network backends, the entire system fosters an interactive environment that encourages experimentation.

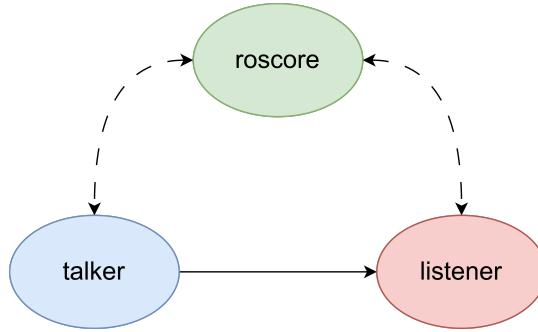


FIGURE 3.3: `roscore` establishes ephemeral connections with the other nodes in the system.

To this point, we have assumed that nodes discover each other, but we have not elaborated on the process. Amidst the extensive network traffic, how do nodes locate and initiate message exchange? The solution lies in a program known as '`roscore`'.

3.3.2 Roscore [58]

`roscore` serves as a vital component within the ROS ecosystem by facilitating connections between nodes to enable message transmission. During initialization, each node registers its published message streams and desired subscriptions with `roscore`, allowing it to establish direct peer-to-peer connections with other nodes participating in the same message topics. A functioning `roscore` is imperative for any ROS system since it serves as a vital reference point for nodes to discover one another.

It's important to note that while `roscore` plays a crucial role in aiding nodes in locating their peers, the actual message transmission between nodes occurs in a peer-to-peer manner. This setup can sometimes be misconstrued, especially for individuals accustomed to client/server systems from web-based backgrounds, wherein the roles of clients and servers are more distinct. The ROS architecture, however, functions as a hybrid system, integrating aspects of both client/server and fully distributed models, thanks to the central role of `roscore`, which acts as a naming service for peer-to-peer message streams.

When a ROS node initiates, it relies on the presence of an environment variable, `ROS_MASTER_URI`, which should contain a URL of the form `http://hostname:11311/`. This URL signifies the existence of a functioning `roscore` accessible on port 11311, hosted on a machine named `hostname`, which can be reached over the network.

With this information, nodes communicate with `roscore` at startup to register themselves and query for other nodes and message streams by name. Each node informs `roscore` about the messages it can provide and those it wishes to subscribe to. `roscore`, in return, supplies the necessary details about the message producers and consumers. In graphical terms, every node within the system can periodically utilize `roscore` services to identify and connect with its peers. This is illustrated by the dashed lines in Figure 3.3, signifying that in a basic two-node setup, the `talker` and `listener` nodes intermittently make service calls to `roscore` while directly engaging in peer-to-peer message exchange.

`Roscore` also serves as a parameter server, extensively utilized by ROS nodes for

configuration purposes. It enables nodes to store and retrieve various data structures, including robot descriptions and algorithm parameters. To interact with the parameter server, ROS provides a command-line tool called '`rosparam`', which we will use throughout this book.

We will delve into examples of using roscore shortly. For now, it's essential to remember that roscore facilitates nodes in discovering other nodes. Before we proceed to run some nodes, it's worth understanding how ROS organizes packages and gaining some insight into the ROS build system, known as 'catkin'. [38]

3.3.3 catkin, Workspaces, and ROS Packages

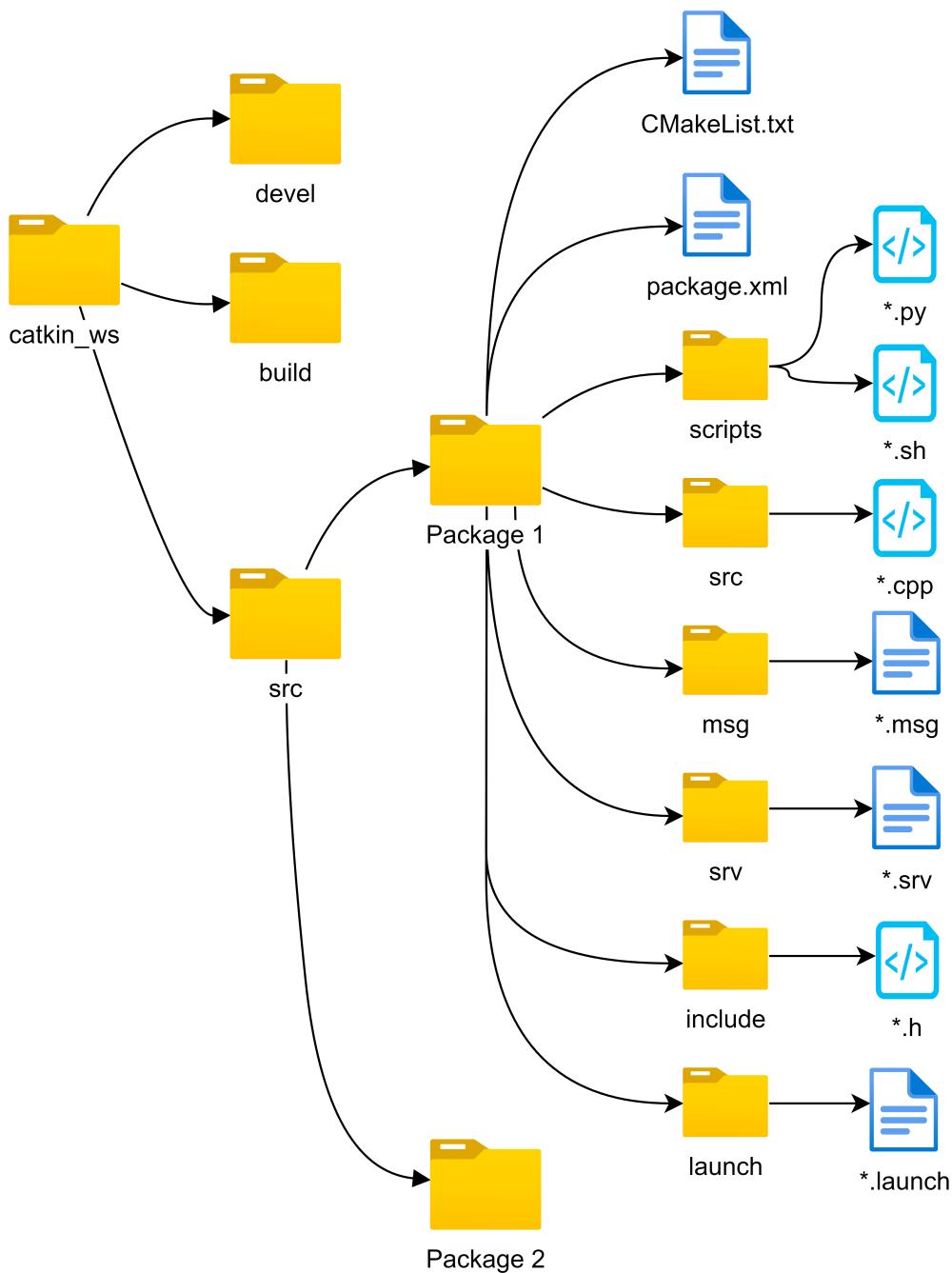


FIGURE 3.4: Files' structure for ROS workspace. [2]

Catkin serves as the ROS build system, comprising a set of tools utilized by ROS for generating executable programs, libraries, scripts, and interfaces that can be employed by other code. If you are developing your ROS code in C++, a good understanding of catkin is essential. However, since this book employs Python for its examples, we won't delve deeply into its intricacies. Nevertheless, we will explore its basic functionalities to some extent. For those interested in a more comprehensive understanding, the [catkin wiki page](#) is an excellent resource. If you are curious about why ROS has its dedicated build system, you can refer to the [catkin conceptual overview wiki page](#). To install ros-melodic catkin workspace, you can use the following command:

Command ➞

```
sudo apt-get install ros-melodic-catkin
```

ROS Catkin Workspace

The ros workspace has several folders as shown in Figure 3.4 above. Following, we will be looking at the function of each folder. [24]

src Folder The 'src' directory within the catkin workspace serves as the designated location for creating or importing new packages from repositories. It's important to note that ROS packages are only built and turned into executables when they reside in the 'src' directory. When the 'catkin_make' command is executed from the workspace directory, it scans the 'src' folder, building each package found there.

build Folder When the 'catkin_make' command is executed within the ROS workspace, the catkin tool generates certain build files and intermediate CMake cache files within the 'build' directory. These cache files play a crucial role in preventing the need to rebuild all packages each time you run 'catkin_make.' For example, if you initially build five packages and subsequently introduce a new package to the 'src' folder, only the new package will be built during the next 'catkin_make' command. This efficiency is achieved through the utilization of cache files within the 'build' directory. It's important to note that deleting the 'build' folder will trigger a complete rebuild of all packages.

devel Folder When 'catkin_make' is executed, it triggers the build process for each package, resulting in the creation of target executables if the build is successful. These executables are saved within the 'devel' folder, which contains shell script files designed to incorporate the current workspace into the ROS workspace path. Access to the packages within the current workspace is only enabled when this script is executed. Typically, the following command is employed for this purpose.

Command ➞

```
source ~/<workspace_name>/devel/setup.bash
```

3.4 Nodes in ROS

Definition ➤

In the context of ROS, a node represents a computational process that undertakes specific tasks. These nodes collaborate within a graph structure and communicate through topics, RPC services, and the Parameter Server. [32]

The architecture encourages the deployment of numerous fine-grained nodes in a robot control system. For instance, individual nodes might be responsible for tasks like managing a laser range-finder, controlling wheel motors, handling localization, executing path planning, providing a graphical system view, and more.

3.4.1 Benefits of Using Nodes

1. **Fault Tolerance:** Nodes enhance fault tolerance by isolating crashes to individual components. If one node fails, it doesn't necessarily affect the entire system.
2. **Reduced Code Complexity:** Compared to monolithic systems, the use of nodes helps reduce code complexity. Each node focuses on specific functionalities, making the system more modular and easier to understand.
3. **Implementation Flexibility:** Nodes expose a minimal API to the overall system, allowing alternative implementations, even in different programming languages, to be seamlessly integrated.

3.4.2 Node Identification

Every running node in ROS possesses a unique graph resource name that distinguishes it within the system. For instance, "/hokuyo_node" might be the identifier for a Hokuyo driver responsible for broadcasting laser scans.

3.4.3 Node Type

Nodes also have a node type, simplifying the referencing of a node executable on the filesystem. Node types are determined by package resource names, incorporating both the node's package and the node executable file's name. When resolving a node type, ROS searches for all executables in the specified package and selects the first one it encounters. It's crucial to avoid producing different executables with the same name within the same package to prevent conflicts.

A ROS node is written with the use of a **ROS client library**, such as `roscpp` or `rospy`.

3.5 ROS Master Communication

The **ROS Master** functions as a provider of naming and registration services for the various **nodes** within the ROS system. It manages the tracking of publishers and subscribers to **topics**, as well as **services**. Essentially, the role of the Master is to facilitate the discovery of individual ROS nodes, allowing them to establish peer-to-peer communication.

Additionally, the Master offers the Parameter Server functionality.

To initiate the Master, the ‘`roscore`’ command is commonly used, which initiates the ROS Master alongside other indispensable components.

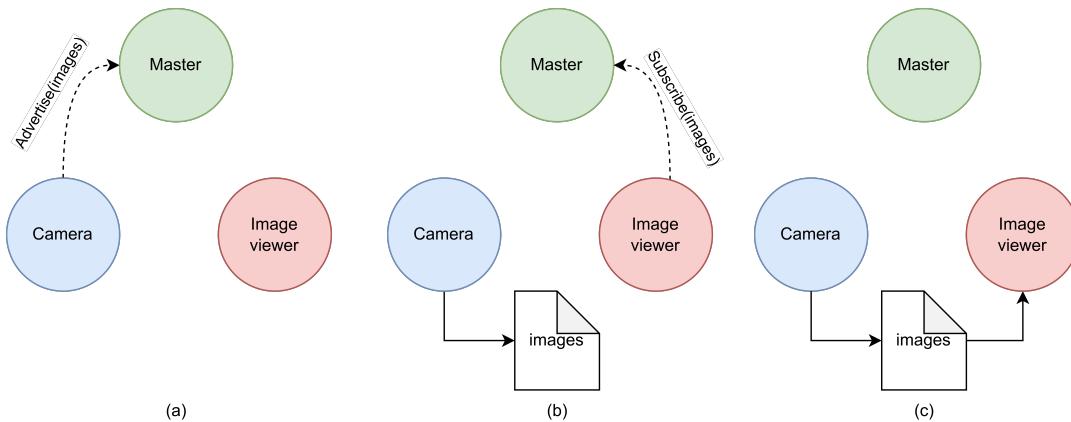


FIGURE 3.5: Master Name Service Example [59]

In the above-shown figure 3.5, let's consider a scenario involving two Nodes: a Camera node and an Image_viewer node. The typical sequence of events begins with the Camera node informing the master that it intends to publish images on the ‘images’ topic, as depicted in (a).

Subsequently, the Camera node commences publishing images to the ‘images’ topic. However, as there are no subscribers to this topic, no data is transmitted. In (b), the Image_viewer expresses its interest in subscribing to the ‘images’ topic to check if any images are available.

With both a publisher and a subscriber now present for the ‘images’ topic, the master node facilitates the awareness of Camera and Image_viewer to each other’s existence, allowing them to commence the exchange of images, as illustrated in part (c) of the figure.

The communication models that are going to be described throughout the upcoming sections are the following:

1. Publisher-Subscriber
2. Server-Client Service
3. Server-Client Action

3.5.1 .bashrc File

It's essential to highlight the significance of the .bashrc file for comprehending the ROS-master communication setup. As discussed in the roscore section (3.3.2), several environment variables, such as ROS_MASTER_URI, are established during roscore initialization, often containing a URL in the format `http://hostname:11311/`, where the hostname represents the IP address of the ROS-master computer. The .bashrc, an abbreviation for **bash** read command, serves as a configuration file for the Bash shell environment. Upon the initiation of an interactive Bash shell session, the .bashrc script file executes, encompassing diverse comments, configurations, **environment variable settings**, and functions aimed at customizing the shell experience and automating tasks. In order to open the .bashrc file in a text editor. For example, if you use nano, we run:

Command ➤➤➤

```
nano ~/.bashrc
```

Alternatively, if you use Vim, run the following command:

Command ➤➤➤

```
vim ~/.bashrc
```

Lastly, we edit the .bashrc file and add the following environmental variables using the **export** keyword. "10.10.10.10" is used in this example, because it's the IP address of the WiFi-communication in Niryo-Ned2. [13]

```
.  
. .  
.  
export ROS_MASTER_URI=http://10.10.10.10:11311/  
export ROS_IP=10.10.10.102
```

3.6 Publishers-Subscribers

One fundamental aspect of communication in ROS involves the process of sending or publishing messages from a node to a designated topic, followed by the counterpart action of listening or subscribing to those messages from another node.

The intricacies of publishing and subscribing are pivotal elements in the ROS framework, necessitating a prior understanding of topics within the system.

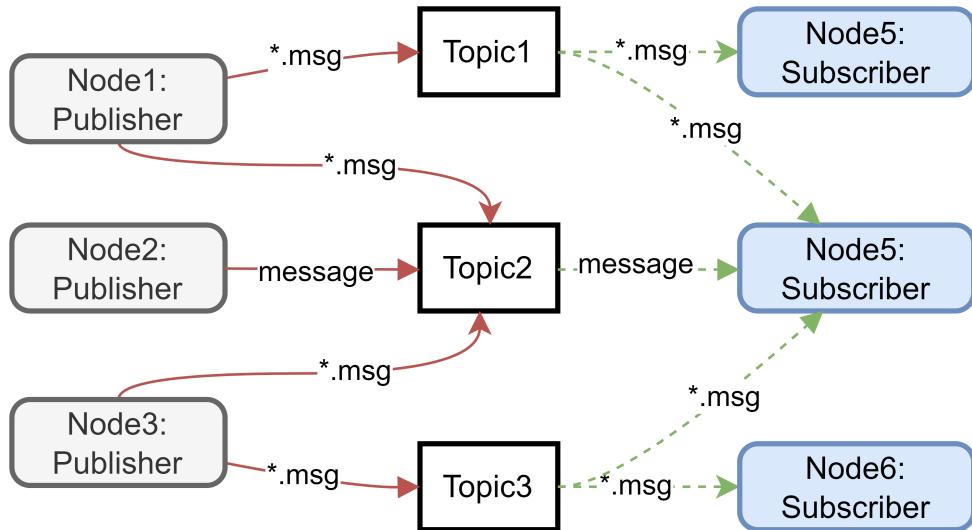


FIGURE 3.6: Publisher–Subscriber communication model.

3.6.1 Topics in ROS

Definition ➤

In ROS, topics serve as named buses, facilitating the exchange of messages between nodes. They operate on anonymous publish/subscribe semantics, allowing the separation of information production from consumption. [12]

Nodes generally remain unaware of the entities they are communicating with. Instead, nodes interested in specific data subscribe to the relevant topic, while nodes generating data publish to that same topic. Multiple publishers and subscribers can exist for a single topic.

Key Characteristics:

- 1. Unidirectional Streaming Communication:** Topics are designed for unidirectional, streaming communication. For remote procedure calls requiring a response, services are recommended, and the Parameter Server is used for maintaining small amounts of state.
- 2. Topic Types:** Each topic is strongly typed by the ROS message type used for publishing. Nodes can only receive messages with a matching type, ensuring consistency in communication. The Master doesn't enforce type consistency among publishers, but subscribers won't establish transport unless the types match.
- 3. Topic Transports:** ROS supports TCP/IP-based (TCPROS) and UDP-based (UDPROS) message transport. TCPROS is the default transport, streaming message data over persistent TCP/IP connections. UDPROS, currently supported only in roscpp, separates messages into UDP packets, making it suitable for low-latency, lossy tasks like teleoperation.

Nodes negotiate the desired transport at runtime, allowing flexibility and future additions.

Tools for Topics: The rostopic command-line tool is available for interacting with ROS topics. For instance, the command:

Command ➞

```
$ rostopic list
```

provides a list of active topics. This tool is valuable for monitoring and managing the flow of information within the ROS system. Whereas, the command line:

Command ➞

```
$ rostopic echo /topic_name
```

will display Messages published to /topic_name. Please refer to the [ros-topic](#) page for more documentation.

3.6.2 Publishing a topic in ROS node

This section shows how to publish a topic in a ROS node. In Python, we use the following syntax with comments to explain each line. The upcoming code examples are taken from [The Construct](#) platform for robotics, please refer to their website or book for a more detailed explanation [53]:

```

1 #! /usr/bin/env python
2
3 # Import the Python library for ROS
4 import rospy
5 # Import the Int32 message from the std_msgs package
6 from std_msgs.msg import Int32
7
8 # Initiate a Node named 'topic_publisher'
9 rospy.init_node('topic_publisher')
10
11 # Create a Publisher object, that will publish on the /counter topic
12 # messages of type Int32
13 pub = rospy.Publisher('/counter', Int32, queue_size=1)
14
15 # Set a publish rate of 2 Hz
16 rate = rospy.Rate(2)
17 # Create a variable of type Int32
18 count = Int32()
19 # Initialize 'count' variable
20 count.data = 0
21
22 # Create a loop that will go until someone stops the program execution
23 while not rospy.is_shutdown():
24     # Publish the message within the 'count' variable
25     pub.publish(count)
```

```

26 # Increment 'count' variable
27 count.data += 1
28 # Make sure the publish rate maintains at 2 Hz
29 rate.sleep()

```

LISTING 3.1: Publisher Node Example

In essence, this code initiates a ROS node and establishes a publisher responsible for continually transmitting a sequence of consecutive integers to the "/counter" topic. To encapsulate:

Definition ➤

A publisher, in ROS terminology, denotes a node fulfilling the role of consistently sending messages to a specified topic. [12]

3.6.3 Messages in ROS

In ROS, communication between nodes is facilitated through topics, with messages serving as the carriers of information. Messages come in various types, and while users have the flexibility to create custom messages, leveraging the extensive collection of default ROS messages is often recommended for efficiency. [53]

Messages are formally defined in .msg files, typically housed within the msg directory of a ROS package. Understanding a message's structure and properties can be accomplished using the `rosmsg show <message>` command, offering insights into the composition of the message and its relevant details.

For example, let's try to get information about the std_msgs/Int32 message. Type the following command and check the output:

Command ➤

```
$ rosmsg show std_msgs/Int32
```

Shell Output:

```
[std_msgs/Int32] :
int32 data
```

For instance, the Int32 message, commonly used in ROS, encapsulates a single variable labeled data with the data type int32. Originating from the std_msgs package, this message definition resides in the msg directory of the package. A closer inspection of the Int32.msg file can be conducted using the following command:

Command ➤

```
$ roscl std_msgs/msg/
```

3.6.4 Subscribing to a topic in ROS node

As shown in Figure 3.6, the publisher nodes are continuously sending messages to a topic/s, and on the other hand, subscriber nodes are listening to a topic or more at a certain point in time.

let's explain a code inside a simple_topic_subscriber.py example script:

```

1 #! /usr/bin/env python
2
3 import rospy
4 from std_msgs.msg import Int32
5
6 # Define a function called 'callback' that receives a parameter
7 def callback(msg): # named 'msg'
8
9     # Print the value 'data' inside the 'msg' parameter
10    print (msg.data)
11
12 # Initiate a Node called 'topic_subscriber'
13 rospy.init_node('topic_subscriber')
14
15 # Create a Subscriber object that will listen to the /counter topic and
16 # will call the 'callback' function each time it reads something
17 # from the topic
18 sub = rospy.Subscriber('/counter', Int32, callback)
19
20 # Create a loop that will keep the program in execution
21 rospy.spin()
```

LISTING 3.2: Subscriber Node Example, ref: [The Construct](#)

In the described scenario, a subscriber node has been effectively implemented to monitor the /counter topic. When receiving data, the node triggers a designated function responsible for printing the message. Initially, the system remained inactive as there were no publishers transmitting data to /counter topic. However, upon executing the rostopic pub command, a message was successfully published to the /counter topic. Consequently, the associated function executed, printing the numeric value to the console. This event was also observable in the output of the rostopic echo command.

3.7 Services

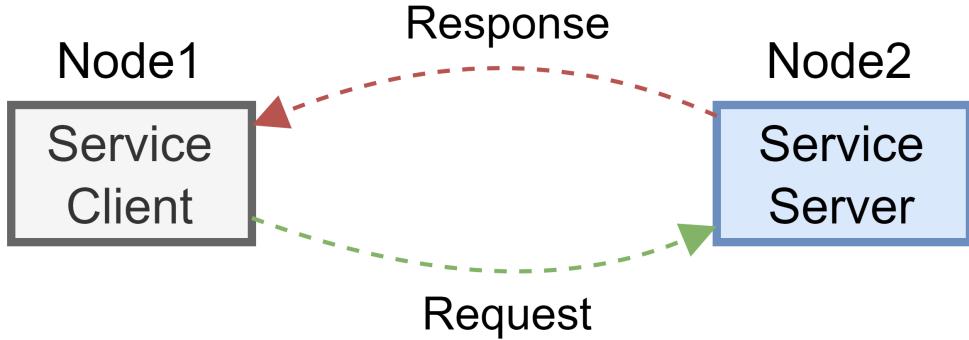


FIGURE 3.7: Server-Client Service communication

3.7.1 Service-Clients node in ROS

```

1 #! /usr/bin/env python
2
3 import rospy
4 # Import the service message used by the service /trajectory_by_name
5 from trajectory_by_name_srv.srv import TrajByName, TrajByNameRequest
6 import sys
7
8 # Initialise a ROS node with the name service_client
9 rospy.init_node('service_client')
10 # Wait for the service client /trajectory_by_name to be running
11 rospy.wait_for_service('/trajectory_by_name')
12 # Create the connection to the service
13 traj_by_name_service = rospy.ServiceProxy('/trajectory_by_name',
14                                         TrajByName)
14 # Create an object of type TrajByNameRequest
15 traj_by_name_object = TrajByNameRequest()
16 # Fill the variable traj_name of this object with the desired value
17 traj_by_name_object.traj_name = "release_food"
18 # Send through the connection the name of the trajectory to be executed
19 # by the robot
20 result = traj_by_name_service(traj_by_name_object)
21 # Print the result given by the service called
22 print(result)

```

LISTING 3.3: Simple Service-Clients node example, ref: [The Construct](#)

The provided Python script is designed to create a service client node in ROS that communicates with a service named '/trajectory_by_name'. Let's break down the script into sequential paragraphs to explain its functionality [53]:

- Shebang and Import Statements:** The script begins with the shebang line specifying the Python interpreter to be used. It then imports the necessary ROS packages, including 'rospy' for interacting with ROS, and the service message classes ('TrajByName' and 'TrajByNameRequest') generated from the 'trajectory_by_name_srv' package.
- Initialize ROS Node:** The first line initializes a ROS node named 'service_client'. A ROS node is a process that performs computation in the ROS environment.

3. **Wait for Service Availability:** The script waits for the service '/trajectory_by_name' to be available before proceeding. This ensures that the service server is up and running.
4. **Create Service Proxy:** A service proxy ('traj_by_name_service') is created to establish a connection to the '/trajectory_by_name service'. This proxy allows the script to call the service as if it were a local function.
5. **Create Service Request Object:** An object ('traj_by_name_object') of type 'TrajByNameRequest' is instantiated. This object is used to store the request data that will be sent to the service.
6. **Set Service Request Parameters:** The 'traj_name' attribute of the request object is set to the desired trajectory name, in this case, "release_food."
7. **Invoke the Service:** The script calls the service using the created service proxy ('traj_by_name_service') and passes the request object ('traj_by_name_object') as an argument. The result of the service call is stored in the 'result' variable.

In summary, this script initializes a ROS node, creates a service client, sends a service request to execute a trajectory named "release_food," and prints the result obtained from the service call.

3.7.2 *.srv Files

In ROS, the specifications for request and response message types in services are encapsulated in .srv files. These files adhere to a specific structure, ensuring clarity and consistency in the definition of ROS services. The following is an example of a .srv file structure:

```
int32 duration      # The time (in seconds)
---                  # Three dashes always between request/s and response.
bool success        # Did it achieve it?
```

3.7.3 Configuration for Custom Service Compilation

For the seamless compilation of custom ROS services, modifications need to be applied to two essential files within the package structure. These files, namely CMakeLists.txt and package.xml, should be edited in a manner consistent with the approach taken for topics. This ensures the proper integration and compilation of custom services in the ROS package. [22]

Modification of CMakeLists.txt

To enable the compilation of custom ROS services, four key functions within the CMakeLists.txt file need to be modified. These modifications facilitate the integration of service messages into the ROS package [52]. Below is a breakdown of the necessary changes:

1. `find_package()`:

- This function specifies all the packages required to compile messages of topics, services, and actions.
- The listed packages are for obtaining paths and are not directly imported for compilation.
- The same packages listed here should be included in the package.xml file under 'build_depend'.

```
find_package(catkin REQUIRED COMPONENTS
std_msgs
message_generation
)
```

2. `add_service_files()`:

This function enumerates all the service messages defined in the srv folder of the package.

For example:

```
add_service_files(
FILES
MyCustomServiceMessage.srv
)
```

3. `generate_messages()`:

- In this function, packages needed for the compilation of service messages are imported.
- Dependencies on standard message types, like std_msgs, should be specified.

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

4. `catkin_package()`:

- Here, list all the packages that will be required by anyone executing something from your package.
- All packages listed here must also be included in package.xml under '<exec_depend>'.

```
catkin_package(
CATKIN_DEPENDS rospy
)
```

Modification of package.xml

To configure the package.xml file for the compilation of custom ROS services, specific modifications are required. Follow these steps to incorporate the necessary changes:

1. Open the package.xml file and add the message_generation package as a build dependency. This ensures that the package is available for message compilation during the build process.

```
<build_depend>message_generation</build_depend>
```

2. Include the following lines to specify the runtime dependencies for message generation. These lines ensure that message_runtime is considered both for building and executing programs within the package.

```
<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

At the end of the modifications, your package.xml file should resemble the following:

```

<?xml version="1.0"?>
<package format="2">
<name>my_custom_srv_msg_pkg</name>
<version>0.0.0</version>
<description>my_custom_srv_msg_pkg package</description>

<maintainer email="user@todo.todo">user</maintainer>

<license>TODO</license>

<buildtool_depend>catkin</buildtool_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<build_export_depend>rospy</build_export_depend>
<exec_depend>rospy</exec_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>std_msgs</exec_depend>
<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>

<export>
</export>
</package>

```

Once you're done, compile your package and source the newly generated messages:

Command ➤➤➤

```

$ roscd;cd ..
$ catkin_make
$ source devel/setup.bash

```

3.7.4 Service-Server node in ROS

To implement a custom service server in a ROS package after creating the necessary .srv file, you need to create a Python program that defines the server's behavior. Follow these steps to create a service server using the example provided [52]:

1. Create a Python script, let's name it 'custom_service_server.py'. Make sure it has the necessary shebang line (`#!/usr/bin/env python`) at the beginning.

2. Import the required ROS packages and the message classes generated from your custom service message ('MyCustomServiceMessage' and 'MyCustomServiceMessageResponse').
3. Define a callback function ('my_callback') that will be executed when the service is called. The request message is passed as an argument, and you can access its fields (e.g., 'request.duration'). Create a response message ('my_response') based on the received request. [21]
4. Initialize the ROS node, and create a service named '/my_service' with the specified callback function ('my_callback').
5. Keep the service open and responsive by calling 'rospy.spin()'.

```

1 #! /usr/bin/env python
2
3 import rospy
4 # you import the service message python classes generated from
5 #   MyCustomServiceMessage.srv
6 from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage,
7     MyCustomServiceMessageResponse
8
9
10 def my_callback(request):
11     print("Request Data--> duration="+str(request.duration))
12     my_response = MyCustomServiceMessageResponse()
13     if request.duration > 5.0:
14         my_response.success = True
15     else:
16         my_response.success = False
17     # the service Response class, in this case
18     #   MyCustomServiceMessageResponse
19     return my_response
20
21
22 rospy.init_node('service_client')
23 # create the Service called my_service with the defined callback
24 my_service = rospy.Service('/my_service', MyCustomServiceMessage ,
25                           my_callback)
26 rospy.spin() # maintain the service open.

```

LISTING 3.4: Simple Service-Server node example, ref: [The Construct](#)

Now, when a client sends a request to the '/my_service' service, the 'my_callback' function will be executed, and a response will be generated based on the request's duration.

To run your service server, execute the script:

Command ➞

```
$ rosrun package_name custom_service_server.py
```

This server will be ready to handle service requests from clients. The provided example checks if the duration of the request is greater than 5.0 and sets the success field in the response accordingly. You can customize the 'my_callback' function based on your specific service requirements. [17]

Here is another instance of a script written by me for a vibration motor operating with a service-server model that is designed for implementation during the later stages of a powder dosing procedure:

```

1 #!/usr/bin/env python
2
3 import time
4 import rospy
5
6 from fx_powder_dosage.srv import VibrationMotor, VibrationMotorRequest,
7     VibrationMotorResponse
8 from niryo_robot_rpi.srv import SetDigitalIO, SetDigitalIOResponse
9
10 def digital_out_d04(value=0):
11     """Making a D04 pin ON or OFF.
12
13     param: value (int) -> off for 0, and on for any other value.
14     """
15     service_name = '/niryo_robot_rpi/set_digital_io'
16     rospy.wait_for_service(service_name, 2)
17
18     service_call = rospy.ServiceProxy(service_name, SetDigitalIO)
19     request = SetDigitalIOResponse()
20
21     request.name = 'D04'
22     request.value = bool(value)
23
24     response = service_call(request)
25
26 def callback_fun(req):
27     on_off = req.on_off # 1 for ON, 0 for OFF
28     freq = req.freq # The frequency [Hz]
29     duty_cyc = req.duty_cycle # [%]
30     time_on = req.time_on # 0 if you want the motor to run until you
31     shut it off. [s]
32     period = 1/freq
33
34     start = time.time()
35     end = 0
36
37     if (time_on > 0) and (on_off == 1):
38         # we need to make this happening for a certain number of second
39         (time_on) !
40         while ((end-start) <= time_on):
41             digital_out_d04(1) # send a high singal to D04
42             time.sleep((duty_cyc/100)*period)
43             digital_out_d04(0) # send a low singal to D04
44             time.sleep((1 - duty_cyc/100)*period)
45             end = time.time()
46
47         print("The duration of the vibration is: {}".format(end-start))
48
49     elif (time_on == 0) and (on_off == 1):
50         digital_out_d04(1)
51     else:
52         digital_out_d04(0)
53
54     response = VibrationMotorResponse(bool(on_off))
55     return response
56
57 def motor_server():
58     rospy.init_node('vibration_server')

```

```

57     s = rospy.Service('vibration_motor', VibrationMotor, callback_fun)
58     print('Server Is called!')
59     rospy.spin()
60
61 if __name__ == "__main__":
62     motor_server()

```

LISTING 3.5: Vibration-Motor Service-Server

From the previous code, we can see that we had 4 requests in this service-server, and one response:

```

int8 on_off
int32 freq
int16 duty_cycle
int16 time_on
---
bool response

```

3.8 Actions

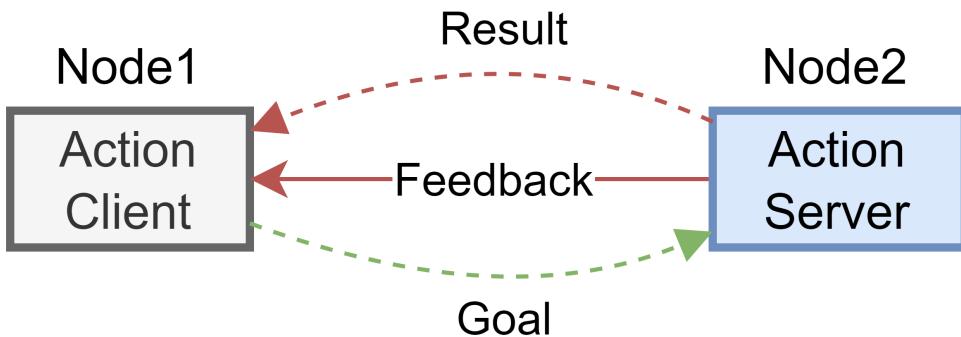


FIGURE 3.8: Server-Client Action communication

Actions represent the most intricate manner of communication in ROS. Leaning on Figure 3.8, an action client node kickstarts the communication by transmitting its destination message (goal) to an action server node. As the dissection at the core commences, the server continues relaying necessary feedback until the task at hand is achieved. Finally, after arriving at the desired goal, data reveals itself as a result back to the client completely its mission. This course builds and depends on the appropriate disposition of three parameters: the goal, feedback, and result message types. Consequently, creating a genuine connection of communications that highlights significant consequences. [16]

```
#goal
PoseStamped pose_goal
float64 wait_duration
---
#result
int32 status
string message
---
#feedback
int32 progression
```

In order for the client and server to interact, we must establish a few messages that they will use to communicate. This is done through an action specification, which outlines the Goal, Feedback, and Result messages that the client and server will exchange. [42]

- **Goal:** When performing actions, we utilize goals that are transmitted from an ActionClient to an ActionServer. To move the base, a PoseStamped message is sent to determine where the robot is to go. Managing the tilting laser scanner requires a goal specifying min and max angles, speed, and other settings.
- **Feedback:** For those who implement servers, feedback offers a method of informing an ActionClient of the gradual advancements of a goal. When moving the base, this could involve the current location of the robot along its route. Managing the angled laser scanner could mean providing information on the instance when the scanning process will conclude.
- **Result:** Once the goal is completed, the ActionServer sends a single result to the ActionClient, distinguishing it from feedback. This feature is extremely helpful for providing desired information needs. When working with a move base, the result's significance is minimal, serving to contain only the robot's final pose. Conversely, the tilting laser scanner's data-packed result could contain the point cloud generated from the requested scan.

3.9 ROS commands summary

There's a method where you can interact with ROS through the terminal window, enabling you to instruct commands into ROS and oversee or govern nodes in a given package using the command prompt. A digestible table 3.1 is available below to access a progression of available commands. [11]

Command	Action	Sample usage & illustration of some subcommands.
roscore	This starts the Master	\$ roscore
rosrun	This runs an executable program and creates nodes	\$ rosrun [package name] [executable name]
rosnode	This shows information about nodes and lists the active nodes	\$ rosnode info [node name] \$ rosnode <subcommand> Subcommand: list
rostopic	This shows information about ROS topics	\$ rostopic <subcommand> <topic name> Subcommands: echo, info, and type
rosmsg	This shows information about the message types	\$ rosmsg <subcommand> [package name]/ [message type] Subcommands: show, type, and list
rosservice	This displays the runtime information about various services and allows the display of messages being sent to a topic	\$ rosservice <subcommand> [service name] Subcommands: args, call, find, info, list, and type
rosparam	This is used to get and set parameters (data) used by nodes	\$ rosparam <subcommand> [parameter] Subcommands: get, set, list, and delete

TABLE 3.1: ROS commands summary with some examples context.

3.10 MoveIt! [47]

MoveIt![7] serves as the primary software framework within the Robot Operating System (ROS) for motion planning and mobile manipulation. It has garnered acclaim for its seamless integration with various robotic platforms, including the PR2 [60], Robonaut [1], and DARPA's Atlas robot. MoveIt! is primarily coded in C++, augmented by Python bindings to facilitate higher-level scripting. Embracing the fundamental principle of software reuse, advocated in the realm of robotics [30], MoveIt! adopts an agnostic approach towards robotic frameworks, such as ROS. This approach entails a formal separation between its core functionality and framework-specific elements, ensuring flexibility and adaptability, especially in inter-component communication.

By default, MoveIt! leverages the core ROS build and messaging systems. To facilitate effortless component swapping, MoveIt! extensively employs plugins across its functionality spectrum. This includes motion planning plugins (currently utilizing OMPL), collision detection (presently incorporating the Fast Collision Library (FCL) [34]), and kinematics plugins (employing the OROCOS Kinematics and Dynamics Library (KDL) [44] for both forward and inverse kinematics, accommodating generic arms alongside custom plugins).

MoveIt!'s principal application domain lies in manipulation, encompassing both stationary and mobile scenarios, across industrial, commercial, and

research settings. For a more comprehensive exploration of MoveIt!, interested readers are encouraged to refer to [7].

Chapter 4

Basics of Powder Dosing

4.1 Affecting Parameters

Part II

Experimental Set-up, Methodology, and Results

Chapter 5

Methodology

In this chapter, we will elucidate the methodology employed in conducting the powder dosage experiments. It will not only encompass crucial Python code with MoveIt motion planning algorithms but also outline the comprehensive sequence logic governing the entire experiment. Additionally, we'll clarify significant functions utilized in constructing the sequence logic and algorithm before delving into the algorithm itself to ensure better comprehension.

5.1 Sequence Logic

In industrial automation, sequence logic refers to a meticulously arranged series of actions orchestrated to accomplish a particular outcome in a methodical and organized manner. These sequences serve as structured procedures executed by automated systems to achieve defined objectives. [9]

The sequential logic can be outlined in bullet points as follows:

1. **Start:** Calibration involves the automatic definition of the Zero positions for each axis of the robot.
2. Retrieve the Pre-saved trajectories.
3. Balance Setup.
4. Obtain the glass crucible.
5. Configure the dosing parameters.
6. Control the powder dosing process while receiving feedback from the balance.
7. Return the crucible upon completion of the dosing process.
8. Unset the balance, repositioning the metal crucible.
9. **End:** Return to the home position.

Each of these sequences will be explained in detail in the state diagram section 5.3.

5.2 Important Functions from FX_ROS Package

The FX_ROS.py package, developed by me, is dedicated to performing all the computational tasks related to motion planning using MoveIt and handling various ROS communication operations. This package efficiently manages operations such as subscribing to specific topics and initializing a ROS service client to invoke service servers for designated actions. It centralizes and streamlines these diverse functionalities within a single framework for enhanced convenience and effective ROS system interaction.

For more extensive details about the full script of FX_ROS package, please refer to Appendix B.

Function	Description	Parameters
Call_Aservice()	Making a service client and sends a request to the server.	service_name, type, request_name, req_args
Subscribe()	Subscribe to a certain ros topic, and wait return the published message	topic_name, type, msg_args
Get_joints()	Subscribe to the '/joint_states' topic and return the current joint values	None
get_pose()	Subscribe to the '/robot_state' topic and return the current pose values	None
move_to_joints()	Move the cobot to a given joints values	joints, arm_speed
Move_joint_axis()	Move one joint axis at a time, either to a new position or shift the current position	axis, new, add, arm_speed
Move_to_pose()	Move the cobot to a given pose values	pose_values, arm_speed
Move_pose_axis()	Move one pose axis at a time, either to a new position or shift the current position	axis, new, add, arm_speed
FK_Moveit()	Make use of MoveIt!'s service-server to calculate and obtain the pose values based on the given joint values	joints

TABLE 5.1: Most important functions used in the research

This table showcases the key functions, listed in table 5.1. Each function name is clickable, leading directly to its function script. Alternatively, see Appendix A for comprehensive descriptions of each function. Worth mentioning, that the robot's movements are primarily planned and calculated by the functions in green, with a heavy reliance on joints over pose coordination. The intent was to obviate the need for any inverse kinematics calculations. Be that as it may, there are instances in which pose coordinations are still brought into play.

5.3 State Diagram

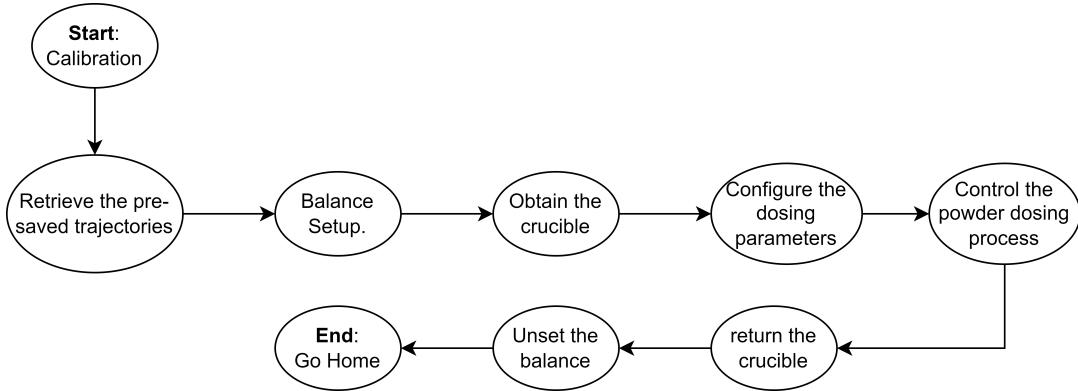


FIGURE 5.1: The sequence logic describing the full dosing process

The state diagram, resembling a state machine, comprises states, transitions, events, and activities. While the activity diagram elucidates the control flow between various activities across the involved objects, the state diagram demonstrates the control flow from one state to another within a singular object. It defines the progression of states an object experiences in reaction to events, including the corresponding responses to those events within the context of the sequence logic. [33]

In this section, we are going to understand more about each state in our sequence logic.

5.3.1 Zero State

In preparation for the experiments, we must first initialize the connection between ROS-master, represented by our Ned2 cobot, and the client, our Linux Ubuntu 18.04 OS laptop. To establish this connection, we take the following steps:

1. The acquisition of WiFi access to Ned2 [13] is necessary and can be established through the specifically assigned IP address of 10.10.10.10.
2. Through SSH or a Secure Shell connection [61], connect with Ned2 through the Linux laptop, we initiate the connection by issuing the command:

Command ➞

```
$ ssh niryo@10.10.10.10
$ niryo@10.10.10.10's password: robotics
```

3. Ensure that you modify the .bashrc file of both devices and assign the value of `http://10.10.10.10:11311` to `ROS_MASTER_URI`, referring to chapter 3, section 3.5.1. Additionally, you should set two environmental variables for both devices in the following way:

```
export ROS_MASTER_URI=http://10.10.10.10:11311/
export ROS_IP=Device_IP
```

4. In order to shut down the roscore and reboot the launch nodes for recognition of the new setup of ros-master defined in the .bashrc file, one must eliminate all launch nodes. Follow the given three commands to achieve this:

Command ➤➤➤

```
$ source ~/.bashrc
$ sudo service niryo_robot_ros stop
$ roslaunch niryo_robot_bringup niryo_ned2_robot.launch
```

Now the cobot is ready to calibrate and take commands.

5.3.2 Calibration

Ned2 undergoes an automatic calibration process, essential for its control, aligning the position of each motor with its respective controller [13]. The calibration sequence involves the following steps for the initial three axes:

1. The axis rotates in the direction depicted in the accompanying image until it halts.
2. It halts upon detecting the limit.
3. The axis retraces its motion in the opposite direction.
4. The axis rotates towards the limit stop.
5. It halts upon detecting the limit stop, establishing the Zero position at this point.



FIGURE 5.2: Calibration Process

To initiate the calibration of the motors, simply contact the service server at '/niryo_robot/joints_interface/calibrate_motors' and request a value of 1. One way to request service is by using the terminal:

```
$ rosservice call /niryo_robot/joints_interface/calibrate_motors
[TAB] [TAB]
$ rosservice call /niryo_robot/joints_interface/calibrate_motors
"value: 1"
```

Alternatively, invoke the Python API to access the service via the GUI that we created.

```
1 def motor_cal():
2     cal_service = '/niryo_robot/joints_interface/calibrate_motors'
3     Call_Aservice(cal_service, type=SetInt, request_name=SetIntRequest,
req_args={"value":1})
```

5.3.3 Retrieve the Pre-Saved Trajectories

In the following table 5.2, we will briefly describe the trajectories used in our motion planning algorithms to conduct the experiments in this research. Every time we open the GUI to conduct the experiment, we need to load a lookup table named "newbalance.xlsx" which contains all needed positions.

Position Name	Description	Number
pour(n)	Represents the placement of the metallic crucible, with n indicating the position, which can vary between 1 and 4.	4 positions.
pour(n)_out	The same as pour(n), whilst we move 8cm more in the z-axis. Check Fig. 5.4(b) for positions, and Fig. 5.5 for n indexing.	4 positions
balance & above_balance	The positions are needed to place the metallic crucible carefully on top of the balance.	2 positions
pick(n)	Refers to the position of the glass crucible.	4 positions
steady(n)	The position needed to approach the glass crucible from the bottom. Refer to Fig. 5.4(a)	4 positions
pick(n)_out1	1 st position needed to get the glass crucible out of its holder, without intersecting with the frame.	4 positions
pick(n)_out2	Takes the crucible away from the frame to avoid collisions when moving the given angle position.	4 positions
(α)angle	The position of TCP when dosing. Whereas α is representing the angle between the negative x-axis and the end-point of the cobot. Check Fig. 5.3	7 positions
(α)angle_rev	Same as the previous, however, with a different orientation of joint 6. Used specifically for 16°, and 21° for the algorithm to work.	2 positions

TABLE 5.2: The positions used to plan the trajectories.

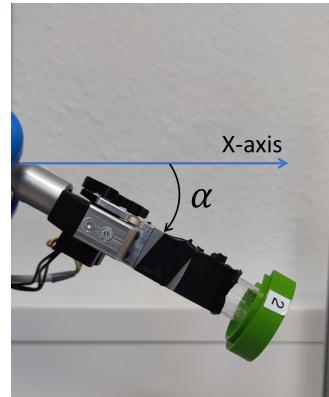


FIGURE 5.3: The angle α with reference to the $x - axis$.

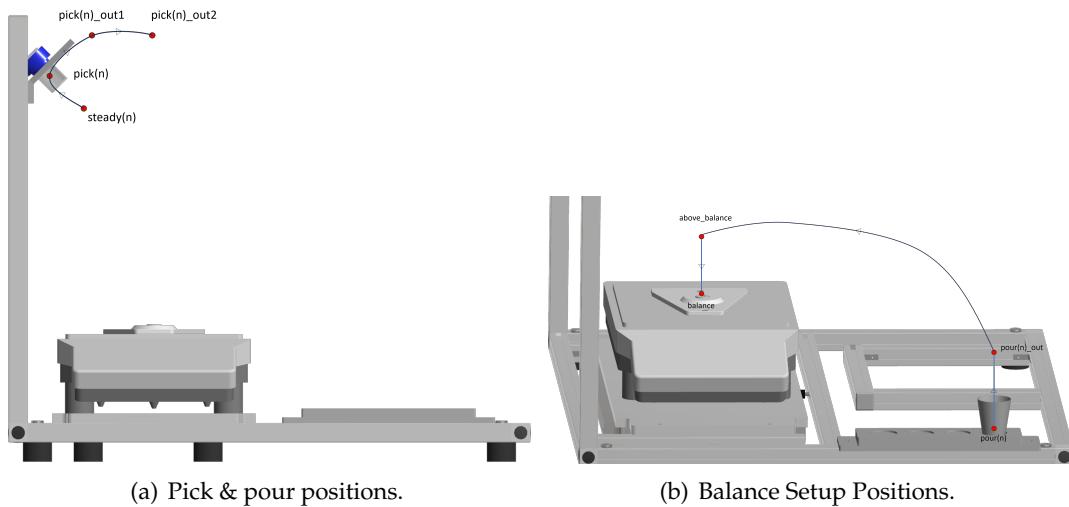


FIGURE 5.4: Defining the pre-named positions used in the trajectory planning.

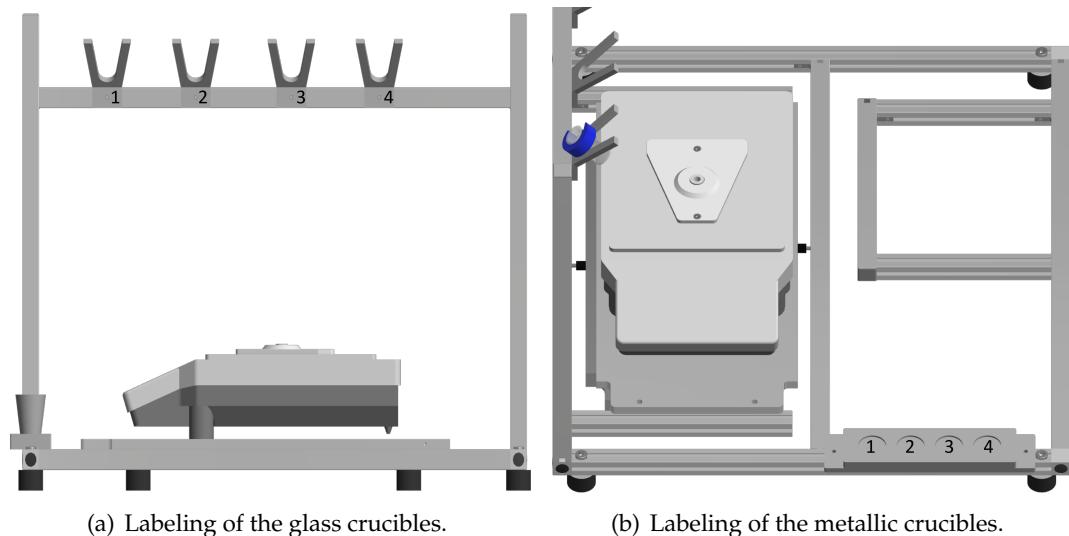


FIGURE 5.5: Indexing the crucibles' holders with numerical values.

5.3.4 Balance Set-Up

In this segment, we'll delve into the process of picking a metallic crucible, positioning it on the balance device, and preparing the balance for the dosing procedure. This step is crucial in ensuring the accuracy and precision of the measurements taken. So, let's explore the finer details of this essential task.

1. Start: Check whether Setup = True?
2. If yes, check whether the input n (the number of crucibles) is in the range from 1 to 4.
3. If yes, Load the needed positions: pour(n), pour(n)_out, balance, above_balance.
4. Open the gripper, and start moving by going to the 'straight forward position'
5. Wait for a second and go to pour_out (above the metallic crucible) and then move slowly towards the pour position.
6. Wait half a second and close the gripper.
7. After picking the crucible, get the arm up by moving back to the pour_out position.
8. Move now to the above_balance position, and slowly start to approach the balance to release the crucible.
9. Release the crucible by opening the gripper and return back to the above_balance position.
10. Set the balance weight to Zero, and set the Setup global variable to be equal to n.
11. End: Return Setup.

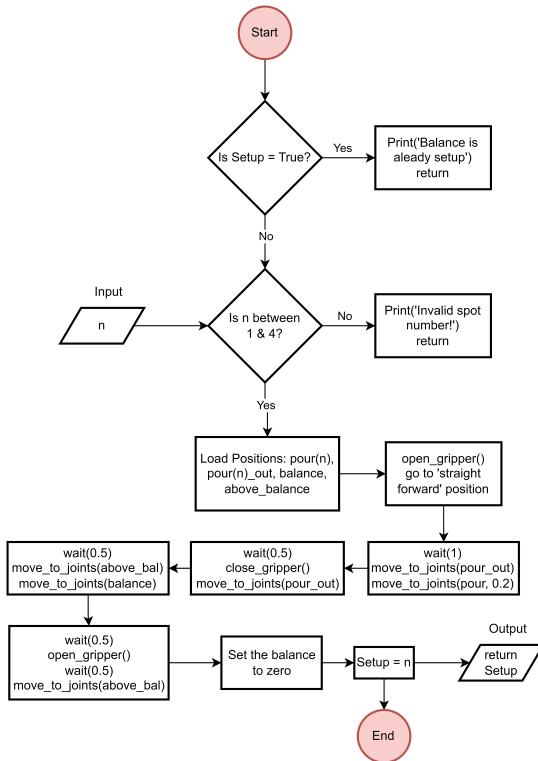


FIGURE 5.6: Balance Setup flowchart

5.3.5 Obtain the Crucible

5.3.6 Do the Experiments

5.3.7 Return the Crucible

5.3.8 Unset the Balance

5.3.9 Go home

Chapter 6

Experimental Set-up

This chapter initiates a comprehensive exploration of the experimental setup. It commences with an overview of the workspace frame, encompassing the array of interconnected devices. These include various crucibles, the balance device, both its hardware and software components, the Ned2-cobot with its hardware configurations and software tools, precision validation procedures and concludes with an examination of the vibration motor, which is an auxiliary component integrated with the cobot.

6.1 Frame

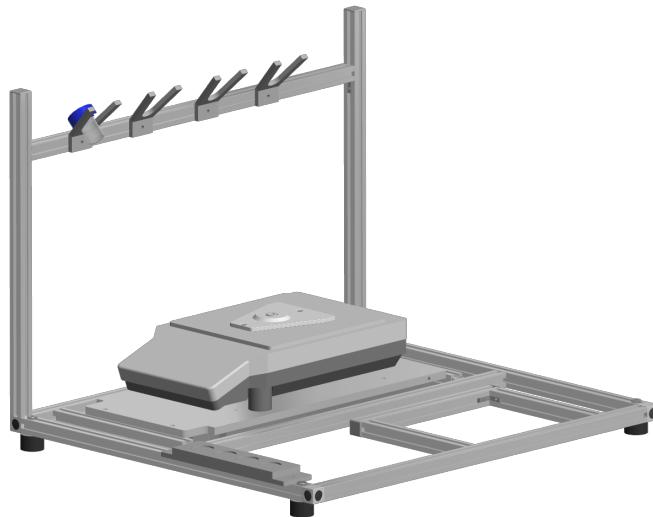


FIGURE 6.1: The Full Framework of System

In the realm of robotics, establishing a meticulously structured workspace framework is of paramount significance. This framework serves as the foundation encompassing all the requisite components with which the cobot will interact. As illustrated in Figure 6.1, this framework has been thoughtfully designed to meet these specifications.

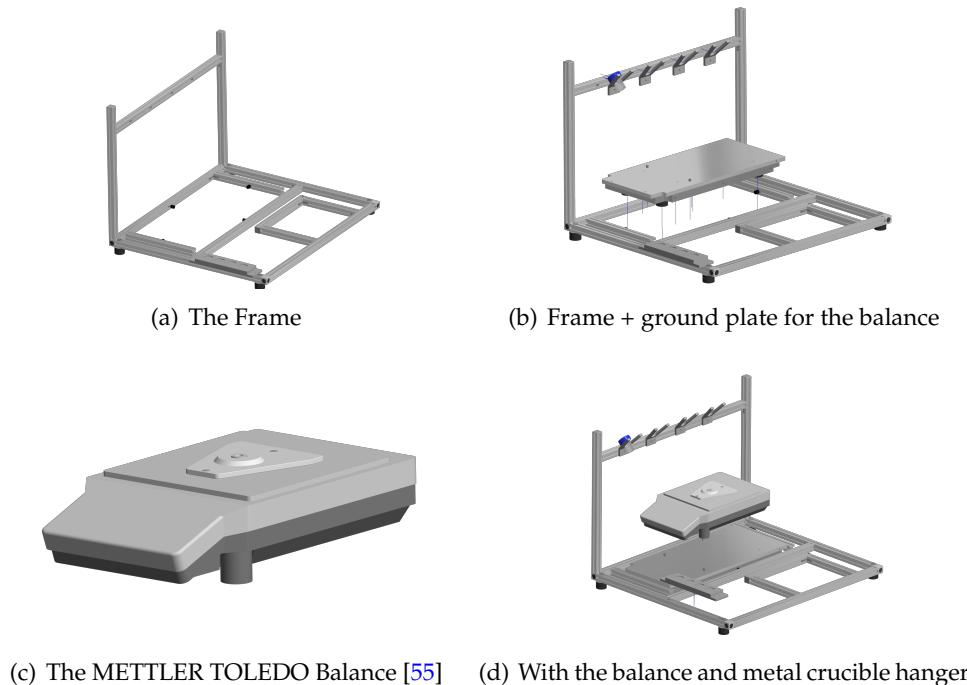


FIGURE 6.2: Demonstration of the parts of the framework.

The framework encompasses various crucial components, as detailed below:

1. **The Frame** itself, depicted in part (a) of Figure 6.2. More comprehensive technical documentation of the frame's constituent parts and dimensions can be found in Appendix F.
2. **The Ground Plate of the Balance**, featuring a rubber base, is illustrated in part (b) of Figure 6.2. This element plays a pivotal role in ensuring the stability of the balance. Given its high sensitivity and precision, it is imperative to eliminate any potential transmission of motion or vibration.
3. The **Mettler Toldedo Me Balance**, showcased in part (c) of Figure 6.2. For an in-depth understanding of the balance's specifications pertinent to our project, please refer to section 6.3.
4. **Four Glass Crucibles** and their corresponding hangers, documented comprehensively in section 6.2.
5. **Four Metal Crucibles** along with their hangers, as outlined in section 6.2.
6. The **Niryo Ned2 Cobot**, expounded upon in detail in section 6.4.
7. **Standing Rubber Buffers**, strategically incorporated to enhance the frame's stability.

Subsequent sections will delve into the intricate particulars and documentation of these essential components for our project.

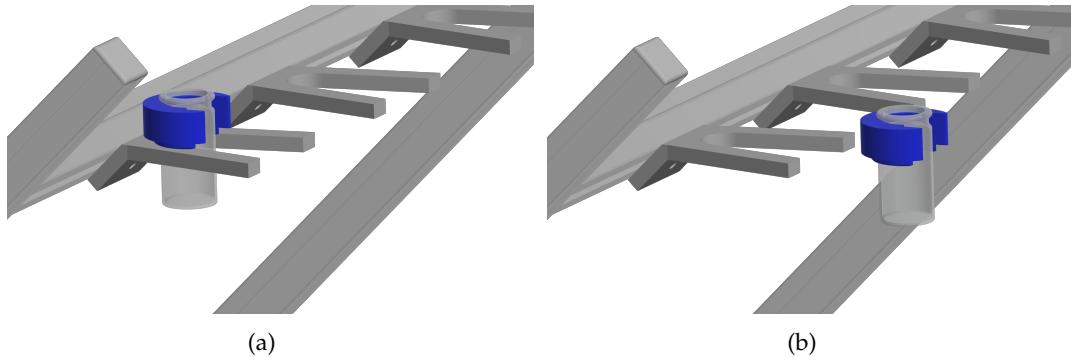


FIGURE 6.3: The glass crucible with its hangers.



(a) The technical design.

(b) An actual crucible on the hanger.

FIGURE 6.4: The metal crucible and its hanger.

6.2 Crucibles

The crucibles play a central role in our project by serving as containers for the powder to be transferred from one crucible to another.

We will utilize two distinct types of crucibles:

- **Glass Crucibles:** These crucibles will predominantly be employed for powder dosing purposes.

In the frame, we have incorporated four hangers for the glass crucibles, which can be observed in Figure 6.1. Additionally, the range of motion for the glass crucible is depicted in Figure 6.3. The crucible has an outer diameter of 22mm, and an inner diameter of about 17mm. More comprehensive information regarding the dimensions and technical specifications of the glass crucible can be found in Appendix E.

- **Metal Crucibles:** These crucibles will hold the powder after it has been dosed from the glass crucibles and will be utilized to measure the weight on the balance.

The hanger serves the purpose of accommodating four crucibles, and you can observe its design in Figure 6.4. It possesses an upper diameter of 37mm and a lower diameter of 25mm, as visualized in Figure 6.4(a). For a more in-depth exploration of the technical design and drawings of this component, please consult the details provided in Appendix E, where comprehensive information is available.

6.3 Balance Device



FIGURE 6.5: The balance with its metal plates and rubber buffers.

The precision balance employed in this study is a product of **METTLER TOLEDO**, a reputable German company. It offers an impressive readability of 0.1mg and a substantial weight capacity, approximately 220g, as referenced in [55]. For comprehensive insights into the specific components and technical specifications of this balance, please consult the information provided in Appendix D. The balance's operation, illustrated in Figure 6.5, involves its integration with two robust metal ground plates and the inclusion of four standing rubber buffers, which collectively contribute to the system's ensured stability during the weighing process.

6.4 Ned2 Collaborative Robot

Ned2 is a collaborative robot, often referred to as a cobot, developed by the French company Niryo [14]. This particular cobot has been purpose-built for educational and research applications, serving as a valuable tool for the development of proof of concepts and experimental work. In the context of this research, the Ned2 cobot played a pivotal role in conducting experiments.

The forthcoming sections delve into an in-depth examination of the hardware specifications and software options offered by the Ned2 cobot.



FIGURE 6.6: Ned2 cobot provided by Niryo [14].

6.4.1 Hardware Configurations

Ned2 is a six-axis collaborative robot, based on open-source technologies. It is intended for education, research and Industry 4.0." [13]

Incorporating the same aluminum framework as its predecessor, Ned2 maintains its commitment to meeting your exacting standards in terms of durability, precision, and repeatability (with an accuracy, and a repeatability of 0.5 mm).

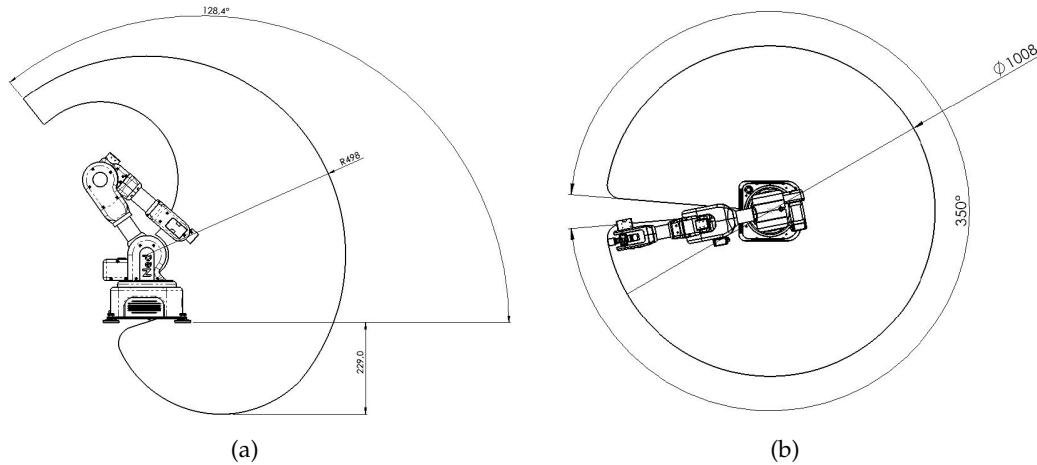


FIGURE 6.8: The detailed workspace of Ned2 cobot.

Ned2 operates on the Ubuntu 18.04 platform and utilizes the ROS Melodic framework, capitalizing on the capabilities of the **Raspberry Pi 4**. This high-performance **64-bit ARM V8 processor**, coupled with **4GB of RAM**, empowers Ned2 to deliver enhanced performance.

This iteration of Ned2 introduces advanced servo motors equipped with Silent Stepper Technology, significantly reducing the operational noise of the robot. As a six-axis robot, the device comprises a total of six distinct motors. The initial three motors are specifically customized and operate as silent Stepper motors, delivering 65 N – cm stall torque. These are equipped with a custom-made control card developed by Niryo. The remaining three motors are servos, with two being XL430 type motors delivering 140 N – cm stall torque, and one XL330 type motor providing 65 N – cm stall torque, as shown in Figure 6.7 [13]. The technical specifications of Ned2 are described as shown in table 6.1 below.

Furthermore, it's crucial for each robot to operate within a well-defined workspace that allows for unrestricted movement. In our case, the workspace of Ned2, as depicted in Figure 5.4 (a) and (b), has been meticulously documented in accordance with Niryo company specifications [13]. Our frame was meticulously designed and engineered to align with the defined workspace of the cobot.

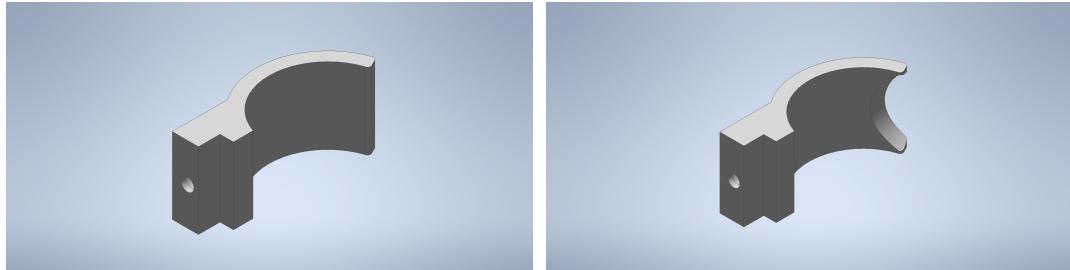


FIGURE 6.7: The Servo motors used in Ned2.

Parameters	Value
Weight (kg)	7
Payload (g)	300
Reach (mm)	440
Degree of freedom	6 rotating joints
Joints range (rad)	$2,949 \leq Joint1 \leq 2,949$ $-2,09 \leq Joint2 \leq 0,61$ $-1.34 \leq Joint3 \leq 1,57$ $-2,089 \leq Joint4 \leq 2,089$ $-1,919 \leq Joint5 \leq 1.922$ $-2,53 \leq Joint6 \leq -2,53$
Joints speed limit (rad/s)	$Joint1 \leq 0.785$ $Joint2 \leq 0.5235$ $Joint3 \leq 0.785$ $Joint4 \leq 1.57$ $Joint5 \leq 1.57$ $Joint6 \leq 1.775$
TCP max speed (mm/s)	468
Repeatability (mm)	+/- 0,5
Footprint (mm)	200x200
Power supply	Input: AC100-240V / 50-60Hz, 2,5A Output: DC 12V - 7A ; 5V - 7A
I/O power supply	5V
Inputs/Outputs Control panel	Digital input x1 Digital output x1
Robot interface	USB2.0 x2 USB3.0 x2 ETHERNET GIGABIT x1
Communication	Modbus TCP (master) TCP/IP
Materials	Aluminum ABS-PC (injection moulding)
Collision detection	Accelerometer & gyroscope in the control panel
Certification	CE Conformity

TABLE 6.1: Technical Specification of Ned2 [13]

Cobot's Gripper

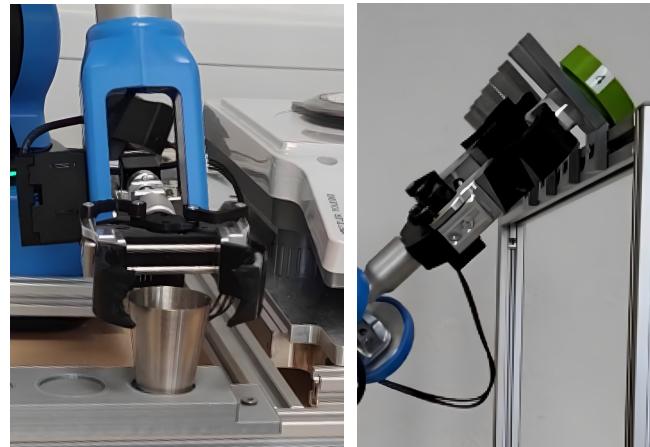


(a) The Initial phase of developing a gripper for the metal crucible only.

(b) The final phase of developing the gripper for both crucibles used in the research.

FIGURE 6.9: The development phase of the gripper.

While it's common to use general-purpose grippers such as those available from Niryo company, our research and development application necessitated a customized gripper. We have engineered a specialized gripper tailored to the unique demands of our project. In Figure 6.9, we can see the development phase of the gripper in order to fit the application for both glass and metal crucibles. The main tasks that the gripper is going to be used are holding the metal crucible from the middle, as shown in part (a) Figure 6.10, and holding the glass crucible from the bottom by the gripper's tip, as shown in part (b) Figure 6.10



(a) The initial task. (b) The second task.

FIGURE 6.10: The main tasks of the gripper.

6.4.2 Software Tools

Ned2 represents a collaborative robot, hinging on the Ubuntu 18.04 platform and **ROS Melodic**—a widely adopted open-source solution in the field of robotics. Leveraging ROS, Ned2 offers an extensive array of libraries that empower users to create a wide spectrum of programs, from the simplest to the most intricate, thus ensuring adaptability to diverse operational requirements [13].



FIGURE 6.11: Software tools supported by Niryo in Ned2. [49]

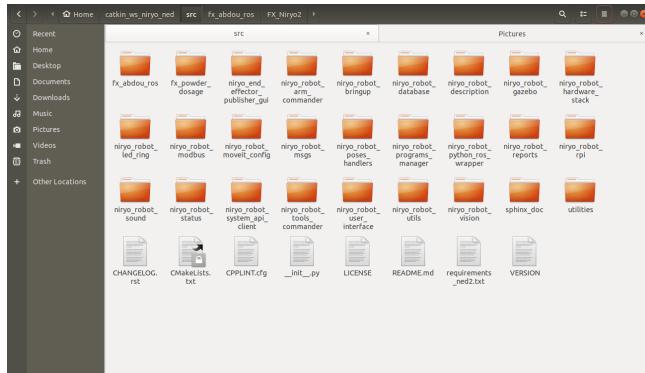


FIGURE 6.12: The Catkin workspace in Ned2

The software tools available for connecting and controlling the Niryo Ned2 cobot are extensive and diverse. They primarily include **Niryo Studio**, an interface offering multiple connectivity options such as hotspot, Wi-Fi, and Ethernet modes, each prompting calibration. This interface allows users to configure workspaces, adjust camera parameters, and control the conveyor belt. The cobot operates on **ROS**, featuring a range of proprietary and ready-made packages available on **GitHub**. It integrates motion planning capabilities using the MoveIt package, particularly employing the KDL kinematics plugin [44]. Thanks to Tensorflow (open-source machine learning tool developed by Google) [51], and OpenCV (open-source computer vision and machine learning software library) [50] training a computer vision-based model and recognizing objects is possible with Ned2. It supports multiple programming languages, including Blockly, Python, MATLAB [5], and interfaces with Arduino controllers [48], Modbus TCP servers, and various other languages via a TCP/IP server. Moreover, it provides the convenience of running Python scripts directly on the robot using the Python ROS wrapper (PyNiryo), making code execution simpler. Extensive tutorials, online documentation [13], and a dedicated support team is available for in-depth exploration and assistance. [15]

However, This research is conducting using ROS-Melodic version, and communication is done through ROS-master communication, with mainly python scripts.

ROS-Melodic

Since the **ROS Melodic** version is installed and used in the raspberry in Ned2 cobot, we had to use Ubuntu 18.4 on the PC where we connected with the robot through ROS-Master communication. Please refer to section 3.1 for more info about Linux.

Niryo has provided various ROS packages in their workspace, as shown in Figure 6.12 below.

6.5 Precision Validation

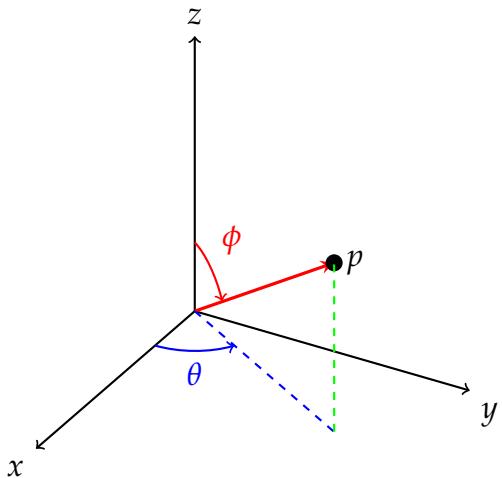
6.6 Vibration Motor

Chapter 7

Evaluation & Results

7.1 Evaluation

This is just a box!



Chapter 8

Conclusion & Future Work

small description of what I have done.. What are my final findings and thoughts...

The future work, what to come.

8.1 Future Work

```
1 #!/usr/bin/env python
2 import rospy
3 from std_msgs.msg import Int32
4 rospy.init_node('topic_publisher')
5 pub = rospy.Publisher('counter', Int32)
6 rate = rospy.Rate(2)
7 count = 0
8
9 while not rospy.is_shutdown():
10     pub.publish(count)
11     count += 1
12     rate.sleep()
```

LISTING 8.1: Some Python code

Appendix A

Important Python Functions Used in this Project

A.1 Calling a Service Server

As shown in the code below, rather than

```

1 def Call_Aservice(service_name, type, request_name=None, req_args=None):
2     """Call a ROS service.
3
4     Parameters:
5     .....
6     service_name: str
7     type: srv
8     request_name: None (srv)
9     req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value':
10      False}
11
12     Returns:
13     .....
14     Return the response of the service.
15     """
16
17     try:
18         rospy.wait_for_service(service_name, 2)
19     except (rospy.ServiceException, rospy.ROSException) as e:
20         rospy.logerr("Timeout and the Service was not available : " +
21 str(e))
22         return RobotState()
23
24     try:
25         service_call = rospy.ServiceProxy(service_name, type)
26         if request_name == None:
27             response = service_call()
28         else:
29             request = request_name()
30             for key, value in req_args.items():
31                 #print("f{key} = {value}")
32                 method = setattr(request, key, value)
33             response = service_call(request)
34     except rospy.ServiceException as e:
35         rospy.logerr("Failed to call the Service: " + str(e))
36         return 0
37
38     return response

```

LISTING A.1: Call_Aservice Function

A.1.1 Example of Call_Aservice Function

The code snippet presented below serves as an illustration of the ease with which the Call_Aservice function is employed. This example specifically functions as a service client aimed at calibrating the motors of the Ned2 robot, a crucial step preceding any motion planning. Refer to section 5.3.2 for comprehensive insights into the motor calibration process.

```

1 def motor_cal():
2     cal_service = '/niryo_robot/joints_interface/calibrate_motors'
3     Call_Aservice(cal_service, type=SetInt, request_name=SetIntRequest,
4                   req_args={"value":1})

```

LISTING A.2: motor_cal Function

A.2 Subscribing to Topic

```

1 def Subscribe(topic_name, type, msg_args):
2     """Subscribe to a certain topic.
3
4     Parameters:
5     .....
6     topic_name: str
7     type: srv
8     msg_args: list >> list of strings, which contains the arguments
9     that we need to read from the topic.
10
11    Returns:
12    .....
13    Return a list of the read values from each argument.
14    If we have only one argument, it returns the value of this argument
15    only, not a list.
16    """
17
18    try:
19        msg = rospy.wait_for_message(topic_name, type, 2)
20    except:
21        rospy.logerr("Timeout and the Topic Did not receive any
22        messages")
23        return 0
24
25    value = []
26    if len(msg_args) == 1:
27        value = getattr(msg, msg_args[0])
28    else:
29        for i in msg_args:
30            value.append(getattr(msg, i))
31
32    return value

```

LISTING A.3: Subscribe Fucntion

A.2.1 Examples of Subscribe Function

The subsequent functions demonstrate the straightforward process of subscribing to a particular topic and retrieving the intended data. In the first case, the topic "/joint_states" acquires its published information concerning the present joint positions from the sensors node. Furthermore, the get_pose

function is designed to subscribe to the "/nirio_robot/robot_state" topic, which disseminates a RobotState message.

```

1 def Get_joints():
2     """return a tuple of 6 values for each joint from 1 till 6"""
3
4     joints_values = Subscribe('/joint_states', JointState, ["position"])
5
6     return joints_values

```

LISTING A.4: Get_joints Function

```

1 def get_pose():
2     """Gets the pose values from the robot_state topic.
3     Return:
4     .....
5     a list of two dictionaries, the first is positions (x,y,z),
6     whereas the second is the rpy (roll, pitch, yaw)
7     """
8
9     return Subscribe('/niryo_robot/robot_state', RobotState, ['position',
10 , 'rpy'])

```

LISTING A.5: get_pose Function

A.3 Moveing the Cobot using Joints

```

1 arm = moveit_commander.move_group.MoveGroupCommander("arm")

1 def move_to_joints(joints, arm_speed=None):
2     """Move to a given joint values.
3     Parameters:
4     .....
5
6     joints: list or tuple -> [joint1, joint2, joint3, joint4, joint5,
7     joint6]
8     arm_speed: float (optional) -> between 0 and 1. (0,1]
9     """
10
11    joints_limits = Get_Joints_limits()
12
13    for i in range(6):
14        if joints_limits.joint_limits[i].max < joints[i] or joints[i] <
15            joints_limits.joint_limits[i].min:
16            print("Joint{} = {}, which is out of limit!".format(i+1,
17                joints[i]))
18            print("Joint{} can not be more than {} neither less than {}"
19                  ".format(i+1, joints_limits.joint_limits[i].max, joints_limits.
20                  joint_limits[i].min))
21            return
22        else:
23            pass
24
25    #arm.set_joint_value_target(joints)
26    if arm_speed:
27        set_speed(arm_speed)
28    arm.go(joints, wait=True)
29
30    arm.stop()

```

LISTING A.6: Moving to a certain joints values

```

1 def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
2     """You should either put a value to add or new, not both.
3
4     Parameters:
5     .....
6     * axis: int -> the number of the joint that you want to move
7
8     * new: float -> The new coordination you want to give to a joint (axis).
9         "new" will always overright the value of the axis.
10    * add: float -> the value in meters change in a certain joint (axis).
11    * arm_speed: float (optional) -> between 0 and 1. (0,1]
12
13    Returns: None
14    .....
15    """
16    moving_joints = list(Get_joints())
17
18    if new:
19        moving_joints[axis-1] = new
20    elif add:
21        moving_joints[axis-1] += add
22
23    joints_limits = Get_Joints_limits()
24
25    if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1]
26    or moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:
27        print("The joint{} can not be more than {} neither less than {}"
28             ".format(axis, joints_limits.joint_limits[axis-1].max,
29             joints_limits.joint_limits[axis-1].min))
30        return 0
31    else:
32        pass
33
34    arm.set_joint_value_target(moving_joints)
35    if arm_speed:
36        set_speed(arm_speed)
37        arm.go(moving_joints, wait=True)
38
39    arm.stop()

```

LISTING A.7: Moving only one axis in joints

A.4 Moveing the Cobot using Pose

```

1 def Move_to_pose(pose_values, arm_speed=None):
2     """Move to a given pose values.
3     Parameters:
4     .....
5
6     pose_values: list or tuple -> [x, y, z, roll, pitch, yaw]
7     arm_speed: float (optional) -> between 0 and 1. (0,1]
8     """
9
10    pose = Pose()
11    p_goal = pose.position
12    orn_goal = pose.orientation
13
14    p_goal.x = pose_values[0]

```

```

15     p_goal.y = pose_values[1]
16     p_goal.z = pose_values[2]
17
18     roll = pose_values[3]
19     pitch = pose_values[4]
20     yaw = pose_values[5]
21
22     orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
23     .quaternion_from_euler(roll,pitch,yaw)
24
25     #arm.set_goal_tolerance(0.001)
26     if arm_speed:
27         set_speed(arm_speed)
28     arm.set_pose_target(pose)
29     arm.go(wait=True)
30
31     arm.stop()
32     arm.clear_pose_targets()

```

LISTING A.8: Moving to certain pose values of the TCP

```

1 def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
2     """You should either put a value to add or new, not both.
3
4     Parameters:
5     .....
6     * axis: str -> (x, y, z, roll, pitch, or yaw)
7     * new: float -> The new coordination you want to give to a certain
8       axis.
9       "new" will always overwrite the value of the axis.
10    * add: float -> the value in meters or radians you want to add to a
11      certain axis.
12    * arm_speed: float (optional) -> between 0 and 1. (0,1]
13    Returns: None
14    .....
15    """
16
17    FK = get_pose()
18    axeses = ['x', 'y', 'z']
19
20
21    pose = Pose()
22    p_goal = pose.position
23    orn_goal = pose.orientation
24
25    p_current = FK[0]
26
27    rpy_current = FK[1]
28
29    if add:
30        if axis.lower() in axeses:
31            current_value = getattr(p_current, axis)
32            setattr(p_current, axis, current_value+add)
33        else:
34            current_value = getattr(rpy_current, axis)
35            setattr(rpy_current, axis, current_value+add)
36
37    if new:
38        if axis.lower() in axeses:
39            setattr(p_current, axis, new)
40        else:
41            setattr(rpy_current, axis, new)
42
43
44    p_goal.x = p_current.x
45    p_goal.y = p_current.y

```

```

41     p_goal.z = p_current.z
42
43     orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
44     .quaternion_from_euler(rpy_current.roll,rpy_current.pitch,
45     rpy_current.yaw)
46
47     arm.set_pose_target(pose)
48
49     if arm_speed:
50         set_speed(arm_speed)
51     arm.go(wait=True)
52
53     arm.stop()
54     arm.clear_pose_targets()

```

LISTING A.9: Moving only one axis in Pose

A.5 Use Forward Kinematics

```

1 def FK_Moveit(joints):
2     """Get Forward Kinematics from the MoveIt service directly after
3     giving joints
4     :param joints
5     :type joints: list of joints values
6     :return: A Pose state object
7     @example of a return
8
9     position:
10    x: 0.278076372862
11    y: 0.101870353599
12    z: 0.425462888681
13    orientation:
14    x: 0.0257527874589
15    y: 0.0122083384395
16    z: 0.175399274203
17    w: 0.984084775322
18
19    """
20    rospy.wait_for_service('compute_fk', 2)
21    moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)
22
23    fk_link = ['base_link', 'tool_link']
24    header = Header(0, rospy.Time.now(), "world")
25    rs = RobotStateMoveIt()
26
27    rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4',
28    'joint_5', 'joint_6']
29    rs.joint_state.position = joints
30
31    reponse = moveit_fk(header, fk_link, rs)
32
33    return reponse.pose_stamped[1].pose

```

LISTING A.10: Using forward kinematics to get the pose coordination
from MoveIt

Appendix B

FX_ROS.py Library in Python

```

1 #!/usr/bin/env python
2 import tf
3 import time
4 import sys
5
6 # ROS client API
7 import rospy
8
9 # ROS services
10 from niryo_robot_arm_commander.srv import GetFK, GetFKRequest,
11     GetJointLimits, JogShift, JogShiftRequest, JogShiftResponse
11 from moveit_msgs.srv import GetPositionFK
12
13 # ROS messages
14 from sensor_msgs.msg import JointState
15 from std_msgs.msg import Header
16 from moveit_msgs.msg import RobotState as RobotStateMoveIt
17 from geometry_msgs.msg import Pose
18 from niryo_robot_msgs.msg import RobotState
19 import moveit_msgs.msg
20
21 import moveit_commander
22 import actionlib
23
24 #robot = moveit_commander.RobotCommander()
25 #scene = moveit_commander.PlanningSceneInterface()
26 #global arm
27 def Connect_to_arm():
28     global arm
29     try:
30         arm = moveit_commander.move_group.MoveGroupCommander("arm")
31     except:
32         raise RuntimeError
33
34 def Call_Aservice(service_name, type, request_name=None, req_args=None)
35     """
36     Parameters:
37     .....
38     service_name: str
39     type: srv
40     request_name: None (srv)
41     req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value': False}
42     should_return ?: None (int) >> is set to 1, if you want to return
43     the response of the service.
44
45     Returns:

```

```

46     .....
47     If should_return is set to 1, the function is going to return the
48     response of the service.
49     Otherwise, the function should only call the service to do a
50     certain action with no return.
51     """
52
53     try:
54         rospy.wait_for_service(service_name, 2)
55     except (rospy.ServiceException, rospy.ROSEException) as e:
56         rospy.logerr("Timeout and the Service was not available : " +
57                     str(e))
57         return RobotState()
58
59     try:
60         service_call = rospy.ServiceProxy(service_name, type)
61
62         if request_name == None:
63             response = service_call()
64         else:
65             request = request_name()
66             for key, value in req_args.items():
67                 #print("f{key} = {value}")
68                 method = setattr(request, key, value)
69             response = service_call(request)
70
71     except rospy.ServiceException as e:
72         rospy.logerr("Failed to call the Service: " + str(e))
73         return 0
74
75     return response
76
77 def Subscribe(topic_name, type, msg_args):
78     """Subscribe to a certain topic.
79
80     Parameters:
81     .....
82     topic_name: str
83     type: srv
84     msg_args: list >> list of strings, which contains the arguments
85     that we need to read from the topic.
86
87     Returns:
88     .....
89     Return a list of the read values from each argument.
90     If we have only one argument, it returns the value of this argument
91     only, not a list.
92     """
93
94     #rospy.init_node('FX_ROS_Subscriber')
95
96     try:
97         msg = rospy.wait_for_message(topic_name, type, 2)
98     except:
99         rospy.logerr("Timeout and the Topic Did not receive any
100 messages")
101         return 0
102
103     value = []
104
105     if len(msg_args) == 1:
106         value = getattr(msg, msg_args[0])
107     else:

```

```

103         for i in msg_args:
104             value.append(getattr(msg, i))
105
106     return value
107
108 def Get_joints():
109     """return a tuple of 6 values for each joint from 1 till 6"""
110
111     joints_values = Subscribe('/joint_states', JointState, ["position"])
112
113     return joints_values
114
115 def get_pose():
116     """Gets the pose values from the robot_state topic.
117     Return:
118     .....
119     a list of two dictionaries, the first is positions (x,y,z),
120     whereas the second is the rpy (roll, pitch, yaw)
121     """
122
123     return Subscribe('/niryo_robot/robot_state', RobotState, ['position',
124     'rpy'])
125
126 def get_pose_list():
127     """Use get_pose() function to get the pose, and turn it into a list
128     .
129     Return:
130     .....
131     A list of floats >>> [x, y, z, roll, pitch, yaw]
132
133     pose = get_pose()
134     position = pose[0]
135     rpy = pose[1]
136
137     return [position.x, position.y, position.z, rpy.roll, rpy.pitch,
138     rpy.yaw]
139
140 def Get_FK_Niryo(joints):
141     """Give the the joints' values to the forward kinematics service
142     provided by Niryo, and get the pose coordinations.
143     """
144     fk_service = '/niryo_robot/kinematics/forward'
145     return Call_Aservice(fk_service, GetFK, GetFKRequest, {'joints': joints}, should_return=1).pose
146
147 def FK_Moveit(joints):
148     """Get Forward Kinematics from the MoveIt service directly after
149     giving joints
150     :param joints
151     :type joints: list of joints values
152     :return: A Pose state object
153     @example of a return
154
155     position:
156         x: 0.278076372862
157         y: 0.101870353599
158         z: 0.425462888681
159     orientation:
160         x: 0.0257527874589
161         y: 0.0122083384395
162         z: 0.175399274203

```

```

160 w: 0.984084775322
161 """
162
163 rospy.wait_for_service('compute_fk', 2)
164 moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)
165
166 fk_link = ['base_link', 'tool_link']
167 header = Header(0, rospy.Time.now(), "world")
168 rs = RobotStateMoveIt()
169
170 rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4',
171 'joint_5', 'joint_6']
172 rs.joint_state.position = joints
173
174 reponse = moveit_fk(header, fk_link, rs)
175
176 return reponse.pose_stamped[1].pose
177
178 def Jog_shift(joints_or_pose, axis, value):
179     """Use the service jog_shift_commander to shift one axis.
180     Parameters:
181     .....
182     joints_or_pose: int >>> 1 for joints_shift, and 2 for pose_shift
183     axis: int >>> (1,2,3,4,5,6) = (x,y,z,roll,pitch,yaw)
184     value: float >> the value for which you want to shift the Jog axis.
185
186     Returns: None
187     .....
188 """
189
190 axis -= 1
191 name = "/niryo_robot/jog_interface/jog_shift_commander"
192 shift_values = [0, 0, 0, 0, 0, 0]
193 shift_values[axis] = value
194
195 req_arg = {'cmd': joints_or_pose, 'shift_values': shift_values}
196
197 Call_Aservice(name, JogShift, JogShiftRequest, req_arg)
198
199 def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
200     """You should either put a value to add or new, not both.
201
202     Parameters:
203     .....
204     * axis: str -> (x, y, z, roll, pitch, or yaw)
205     * new: float -> The new coordination you want to give to a certain
206       axis.
207       "new" will always overwrite the value of the axis.
208     * add: float -> the value in meters or radians you want to add to a
209       certain axis.
210     * arm_speed: float (optional) -> between 0 and 1. (0,1]
211     Returns: None
212     .....
213 """
214
215 FK = get_pose()
216 axes = ['x', 'y', 'z']
217
218 pose = Pose()
219 p_goal = pose.position
220 orn_goal = pose.orientation
221
222 p_current = FK[0]

```

```

220     rpy_current = FK[1]
221
222     if add:
223         if axis.lower() in axes:
224             current_value = getattr(p_current, axis)
225             setattr(p_current, axis, current_value+add)
226         else:
227             current_value = getattr(rpy_current, axis)
228             setattr(rpy_current, axis, current_value+add)
229
230     if new:
231         if axis.lower() in axes:
232             setattr(p_current, axis, new)
233         else:
234             setattr(rpy_current, axis, new)
235
236
237     p_goal.x = p_current.x
238     p_goal.y = p_current.y
239     p_goal.z = p_current.z
240
241     orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
242     .quaternion_from_euler(rpy_current.roll,rpy_current.pitch,
243     rpy_current.yaw)
244
245     arm.set_pose_target(pose)
246
247     if arm_speed:
248         set_speed(arm_speed)
249     arm.go(wait=True)
250
251     arm.stop()
252     arm.clear_pose_targets()
253
254 def Move_to_pose(pose_values, arm_speed=None):
255     """Move to a given pose values.
256     Parameters:
257     .....
258
259     pose_values: list or tuple -> [x, y, z, roll, pitch, yaw]
260     arm_speed: float (optional) -> between 0 and 1. (0,1)
261     """
262
263     pose = Pose()
264     p_goal = pose.position
265     orn_goal = pose.orientation
266
267     p_goal.x = pose_values[0]
268     p_goal.y = pose_values[1]
269     p_goal.z = pose_values[2]
270
271     roll = pose_values[3]
272     pitch = pose_values[4]
273     yaw = pose_values[5]
274
275     orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
276     .quaternion_from_euler(roll,pitch,yaw)
277
278     #arm.set_goal_tolerance(0.001)
279     if arm_speed:
280         set_speed(arm_speed)
281     arm.set_pose_target(pose)
282     arm.go(wait=True)

```

```

280
281     arm.stop()
282     arm.clear_pose_targets()
283
284 def move_to_joints(joints, arm_speed=None):
285     """Move to a given joint values.
286     Parameters:
287     .....
288
289     joints: list or tuple -> [joint1, joint2, joint3, joint4, joint5,
290     joint6]
291     arm_speed: float (optional) -> between 0 and 1. (0,1]
292     """
293
294     joints_limits = Get_Joints_limits()
295
296     for i in range(6):
297         if joints_limits.joint_limits[i].max < joints[i] or joints[i] <
298             joints_limits.joint_limits[i].min:
299             print("Joint{} = {}, which is out of limit!".format(i+1,
300                   joints[i]))
301             print("Joint{} can not be more than {} neither less than {}".
302                 format(i+1, joints_limits.joint_limits[i].max, joints_limits.
303                     joint_limits[i].min))
304             return
305     else:
306         pass
307
308     #arm.set_joint_value_target(joints)
309     if arm_speed:
310         set_speed(arm_speed)
311     arm.go(joints, wait=True)
312
313     arm.stop()
314
315 def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
316     """You should either put a value to add or new, not both.
317
318     Parameters:
319     .....
320     * axis: int -> the number of the joint that you want to move
321
322     * new: float -> The new coordination you want to give to a joint (axis).
323         "new" will always overright the value of the axis.
324     * add: float -> the value in meters change in a certain joint (axis).
325
326     Returns: None
327     .....
328     """
329
330     moving_joints = list(Get_joints())
331
332     if new:
333         moving_joints[axis-1] = new
334     elif add:
335         moving_joints[axis-1] += add
336
337     joints_limits = Get_Joints_limits()
338
339     if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1]
340         or moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:

```

```

334         print("The joint{} can not be more than {} neither less than {}"
335             .format(axis, joints_limits.joint_limits[axis-1].max,
336                     joints_limits.joint_limits[axis-1].min))
337         return 0
338     else:
339         pass
340
341     arm.set_joint_value_target(moving_joints)
342     if arm_speed:
343         set_speed(arm_speed)
344     arm.go(moving_joints, wait=True)
345
346 def Get_Joints_limits():
347     """Getting the limits for each joint.
348
349     You can get any joint limits as following:
350
351     Get_Joints_limits().joint_limits[0 - 5].max (float)
352     Get_Joints_limits().joint_limits[0 - 5].min (float)
353     Get_Joints_limits().joint_limits[0 - 5].name (str)
354
355     Where 0 for (joint 1), and 5 for (joint 6)
356     max, min, or name would give the maximum, minimum, or name of the
357     indicated joint.
358     """
359
360     joints_limits = Call_Aservice('/niryo_robot_arm_commander/
361     get_joints_limit', GetJointLimits)
362     return joints_limits
363
364 def set_speed(speed):
365     """Set a scaling factor for optionally reducing the maximum joint
366     velocity. Allowed values are in (0,1]."""
367     arm.set_max_velocity_scaling_factor(speed)
368
369 def wait(duration):
370     """wait for a certain time.
371
372     :param duration: duration in seconds
373     :type duration: float
374     :rtype: None
375     """
376
377     time.sleep(duration)
378
379 def move_with_action(pose):
380     """Still under development"""
381
382     moveit_commander.roscpp_initialize(sys.argv)
383     rospy.init_node('simple_action', anonymous=True)
384
385     robot_arm = moveit_commander.move_group.MoveGroupCommander("arm")
386
387     robot_client = actionlib.SimpleActionClient('execute_trajectory',
388         moveit_msgs.msg.ExecuteTrajectoryAction)
389     robot_client.wait_for_server()
390     #rospy.loginfo('Execute Trajectory server is available for robot')
391
392     robot_arm.set_pose_target(pose)
393     #robot_arm.set_pose_target([0.29537095654868956, 4.675568598554573e
394     -05, 0.4286678926923855, 0.0017192879795506913,
395     0.0014037282477544944, 0.00016120358136762693])

```

```
389     robot_plan_home = robot_arm.plan()
390
391     robot_goal = moveit_msgs.msg.ExecuteTrajectoryGoal()
392     robot_goal.trajectory = robot_plan_home
393
394     robot_client.send_goal(robot_goal)
395     robot_client.wait_for_result()
396     robot_arm.stop()
397
398 def move_pose_orn(pose, arm_speed=None):
399     """Move to a given pose values, but with orientation not rpy.
400
401     Parameters:
402     .....
403     * pose: A Pose state object
404
405     example of the pose state object that should be given:
406     =====
407     position:
408         x: 0.278076372862
409         y: 0.101870353599
410         z: 0.425462888681
411     orientation:
412         x: 0.0257527874589
413         y: 0.0122083384395
414         z: 0.175399274203
415         w: 0.984084775322
416     =====
417     """
418
419     arm.set_pose_target(pose)
420     if arm_speed:
421         set_speed(arm_speed)
422     arm.go(wait=True)
423
424     arm.stop()
425     arm.clear_pose_targets()
426
427 def move_to_named_pos(position_name, arm_speed=None):
428     """Available names:
429     - 'resting'
430     - 'straight_forward'
431     - 'straight_up'
432     """
433     arm.set_named_target(position_name)
434     if arm_speed:
435         set_speed(arm_speed)
436     arm.go(wait=True)
```

Appendix C

xlsx.py Package in Python

```

1 from openpyxl import Workbook, load_workbook
2 from openpyxl.utils import get_column_letter
3 from openpyxl.styles import Font, Alignment
4
5 positions_dict = {}
6 headers = ['Name', 'Joint_values', 'Pose_values']
7
8 def Create_worksheet(document_name='Untitled'):
9     global positions_dict
10
11     if ".xlsx" != document_name[-5:]:
12         document_name += '.xlsx'
13
14     # Create a workbook object
15     wb = Workbook()
16
17     # Create an active worksheet
18     ws = wb.active
19     ws.title = 'Data'
20
21     for col in range(1, 4):
22         ws[get_column_letter(col) + '1'] = headers [col-1]
23         ws[get_column_letter(col) + '1'].font = Font(bold=True, color="#0099CCFF")
24         ws[get_column_letter(col) + '1'].alignment = Alignment(
25             horizontal='center')
26
27         ws.move_range("C1", rows=0, cols=5)
28         ws.column_dimensions['A'].width = 20
29
30         ws.merge_cells("B1:G1")
31         ws.merge_cells('H1:M1')
32
33     wb.save(document_name)
34     wb.close()
35
36 def Save_position(document_name, position_name, joint_values,
37     pose_values):
38     """Paramters:
39     .....
40     document_name: str
41     position name: str
42     joint_values: list >> list of 6 floats.
43     pose_values: list >> list of 6 floats.
44
45     Returns:
46     .....
47     """
48
49     global positions_dict

```

```

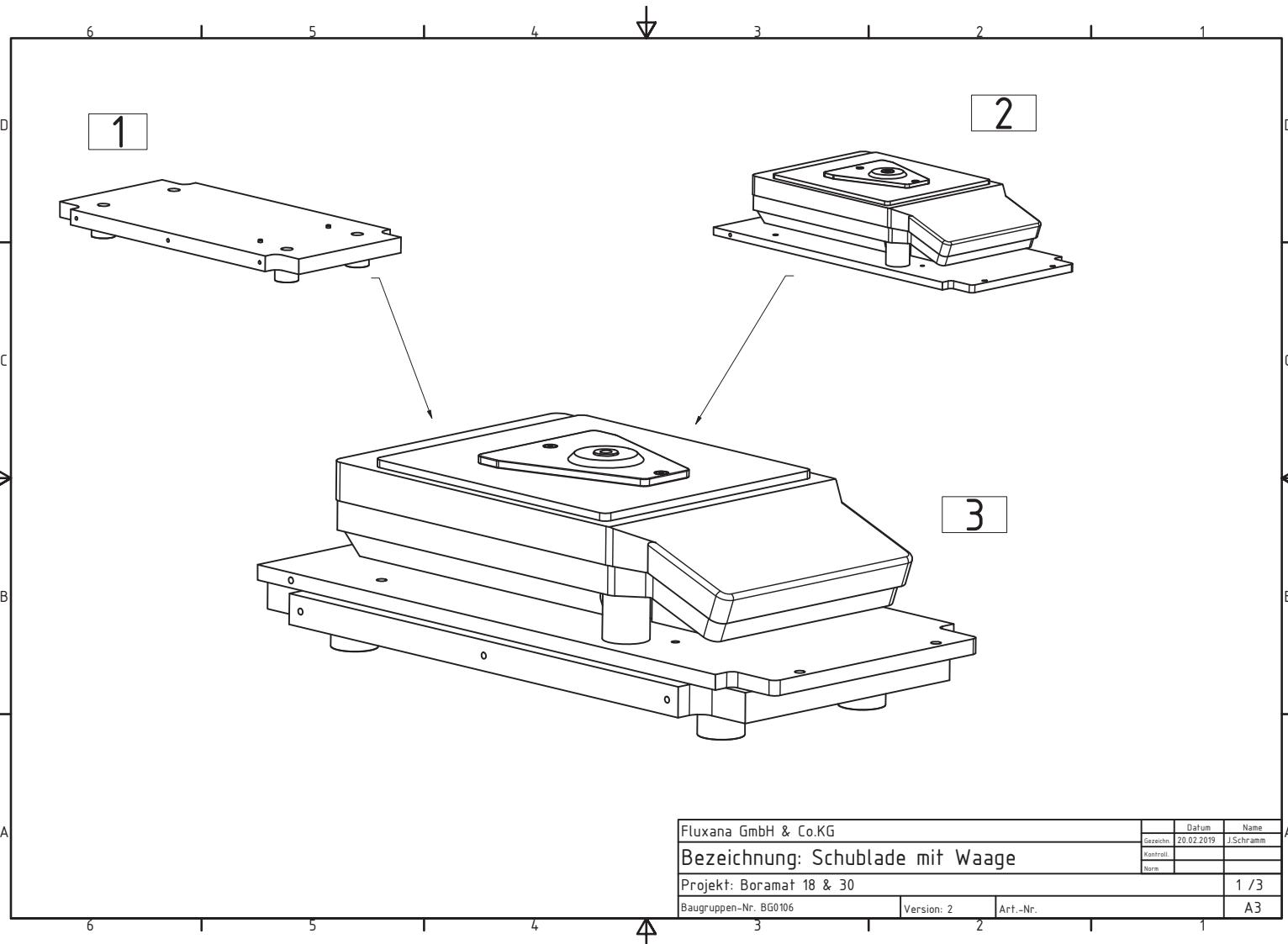
47
48     if ".xlsx" != document_name[-5:]:
49         document_name += '.xlsx'
50
51     wb = load_workbook(document_name)
52     ws = wb.active
53
54     ws.append([position_name] + joint_values + pose_values)
55     wb.save(document_name)
56     wb.close()
57
58 def Load_all_positions(document_name):
59     """Paramter:
60     .....
61     document_name: str --> the name of the excel document you want to
62     load the positions from
63
64     Return:
65     .....
66     position_dict: dictionary --> A dictionary which contains all the
67     positions from the given excel file.
68     The dict has the following structure:
69     {'Position_name': {'Joint_values': [j1,j2,j3,j4,j5,j6], 'Pose_values':
70     ':[x,y,z,roll,pitch,yaw]}}
71     """
72
73     global positions_dict
74
75     if ".xlsx" != document_name[-5:]:
76         document_name += '.xlsx'
77
78     wb = load_workbook(document_name)
79     ws = wb.active
80
81
82     for row in ws.iter_rows(min_row=2, max_col=13):
83         if row[0].value == None:
84             ws.delete_rows(row[0].row)
85             wb.save(document_name)
86         positions_dict[row[0].value] = {}
87         joints = []
88         pose = []
89         for i in range(1,7):
90             joints.append(row[i].value)
91             pose.append(row[i+6].value)
92         positions_dict[row[0].value]['Joint_values'] = joints
93         positions_dict[row[0].value]['Pose_values'] = pose
94
95     wb.close()
96     return positions_dict
97
98 def Load_position(document_name, position_name, debug=True):
99     global positions_dict
100
101     if ".xlsx" != document_name[-5:]:
102         document_name += '.xlsx'
103
104     wb = load_workbook(document_name)
105     ws = wb.active
106
107     for row in ws.iter_rows(min_row=2, min_col=1, max_col=13):
108         if row[0].value == position_name:
109             if position_name in positions_dict.keys():

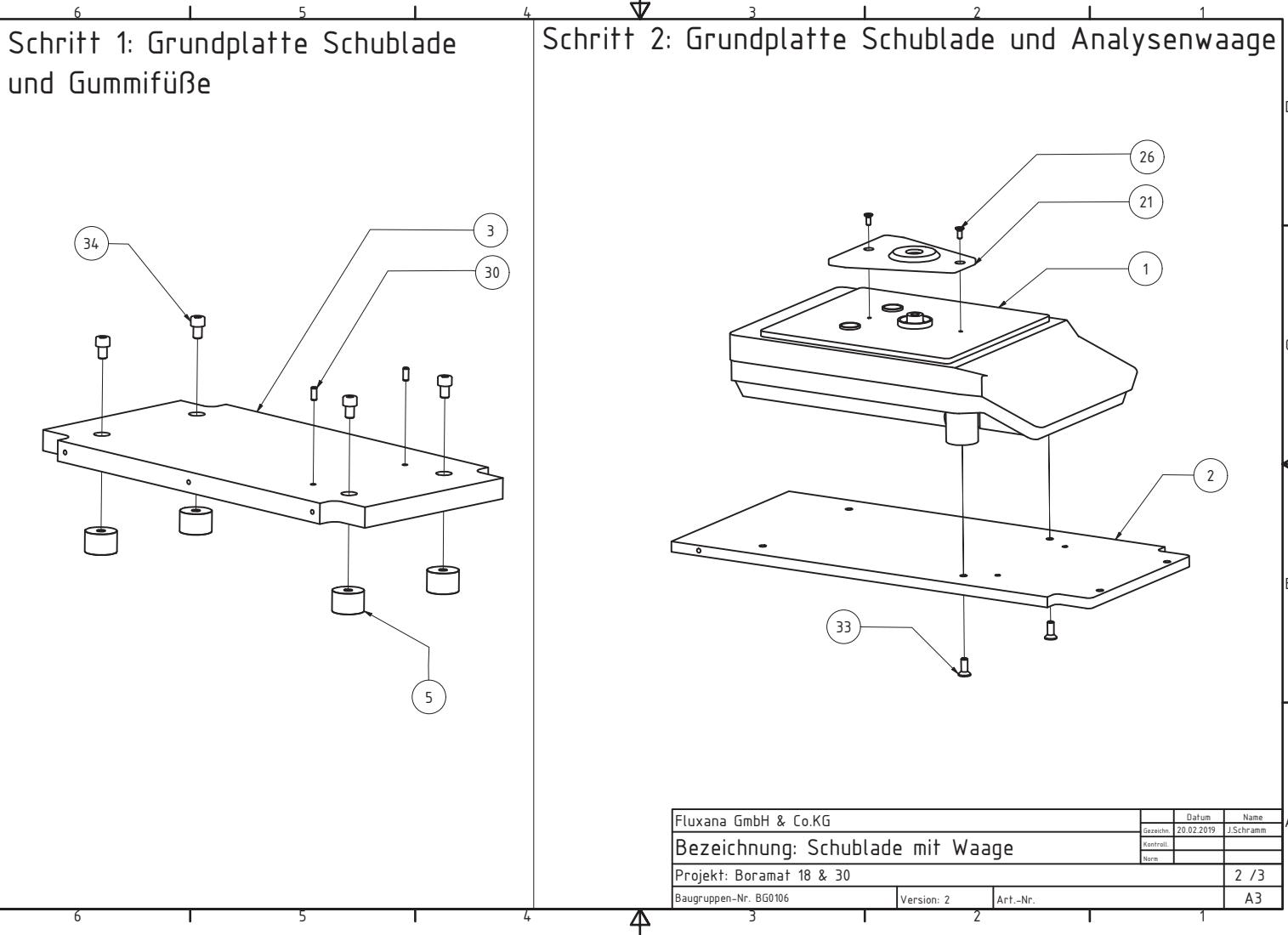
```

```
106             if debug: print("The position is already loaded into  
107     positions dictionary")  
108             wb.close()  
109             return  
110         else:  
111             positions_dict[position_name] = {}  
112             joints = []  
113             pose = []  
114             for i in range(1,7):  
115                 #if row[i].value == None:  
116                 #break  
117                 joints.append(row[i].value)  
118                 pose.append(row[i+6].value)  
119             positions_dict[position_name]['Joint_values'] = joints  
120             positions_dict[position_name]['Pose_values'] = pose  
121  
122             if debug: print('The position is successfully loaded  
123     into the positions dictionary')  
124             wb.close()  
125             return positions_dict  
126  
127     print("Position was not found!")  
128     wb.close()  
129  
130 def delete_pos(document_name, position_name):  
131     if ".xlsx" != document_name[-5:]:  
132         document_name += '.xlsx'  
133  
134     wb = load_workbook(document_name)  
135     ws = wb.active  
136  
137     for row in ws.iter_rows(min_row=2, min_col=1, max_col=1):  
138         if position_name == row[0].value:  
139             ws.delete_rows(row[0].row)  
140             wb.save(document_name)  
141             wb.close()  
142             return  
143  
144     wb.close()  
145     print("No position found with the name: '{}'".format(position_name))
```

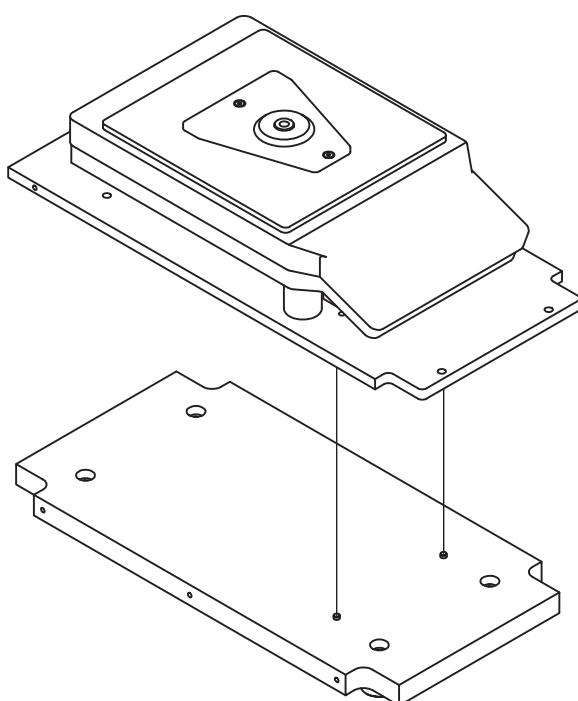

Appendix D

Balance Technical Drawing





Schritt 3: Montage der Führungsschienen



PARTS LIST			
ITEM	QTY	PART NUMBER	DESCRIPTION
1	1		Mettler Toledo Waage
2	1	BT0413	Grundplatte Waage
3	1	BT0414	Grundplatte Schublade
5	4		Gummipuffer 30x20mm M8x8
21	1	BT0419	Schutz Waage
26	2	DIN EN ISO 10642	Senkschraube mit Inbus M4 x 10
30	2	ISO 2338	Zylinderstifte 5 m6 x 12
33	2	DIN EN ISO 10642	Senkschraube mit Inbus M6 x 16
34	4	DIN EN ISO 4762	Zylinderschraube mit Inbus M8 x 12

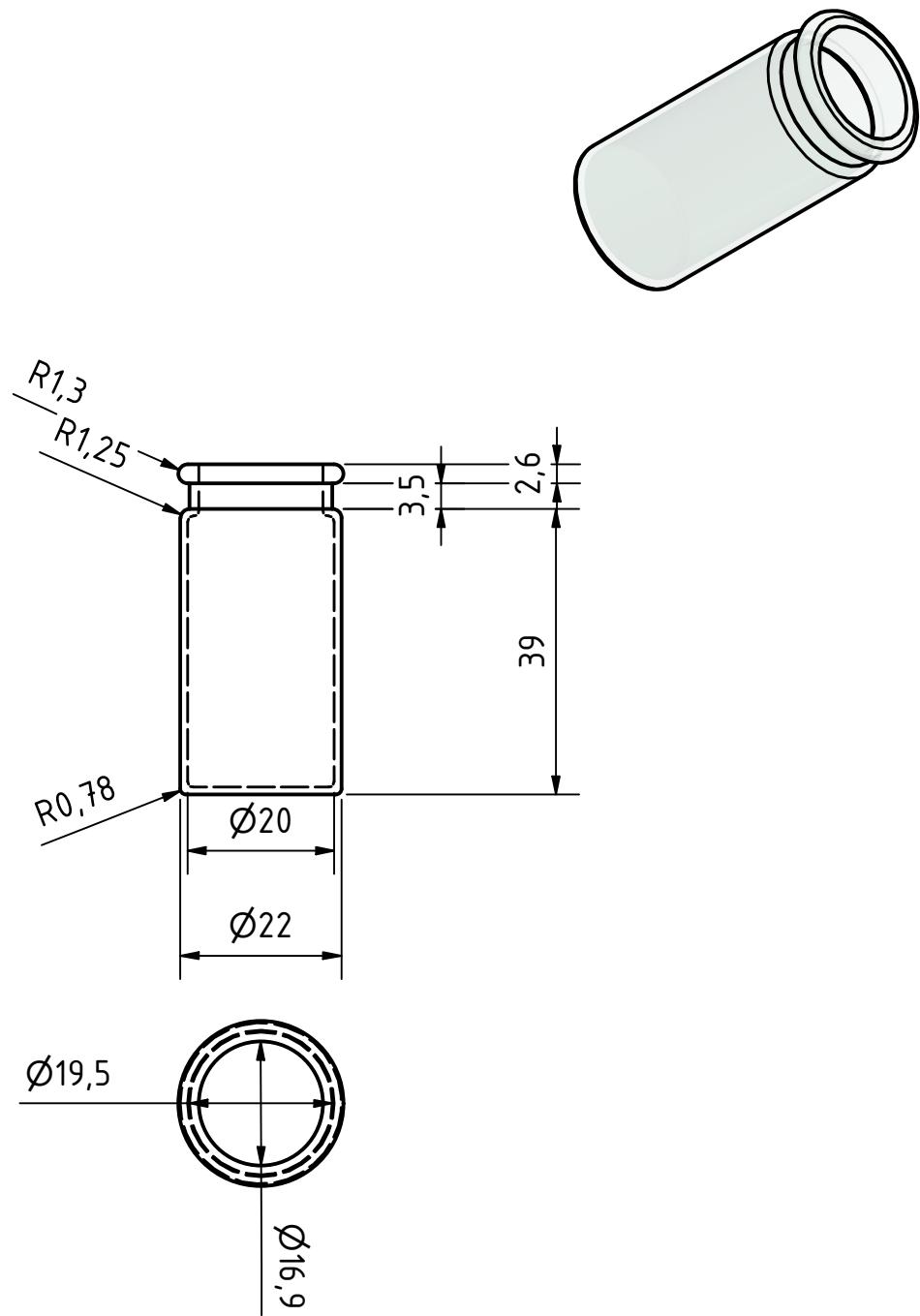
Fluxana GmbH & Co.KG	Datum	Name
Gezeichnet: 20.02.2019	J.Schramm	
Kontrolliert:		
Norm:		
Projekt: Boramat 18 & 30	3 /3	
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr. A3

Appendix E

Glass and Metal Crucibles Technical Drawing

The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksmustereintragung vorbehalten. Schutzvermerk nach ISO 16016 beachten.



Allgemeintoleranz: DIN ISO 2768-mK	Alle Kanten R0,1 - 0,3 entgratet		
------------------------------------	----------------------------------	--	--

Erstellt durch: Abdelrahman Mostafa	Genehmigt von: N/A	Gewicht: -	Werkstoff: Glas
--	-----------------------	---------------	--------------------

Bauteilbezeichnung:

FLUXANA®
XRF Application Solutions

Projekt:

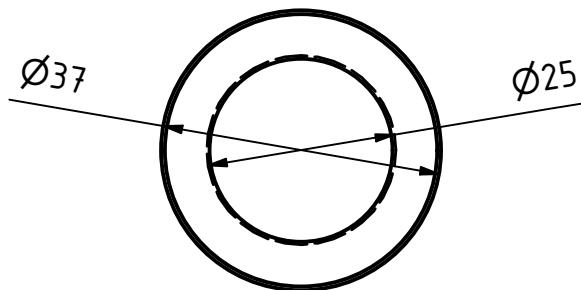
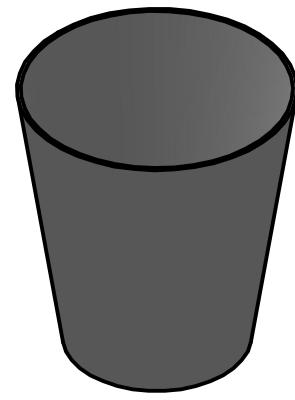
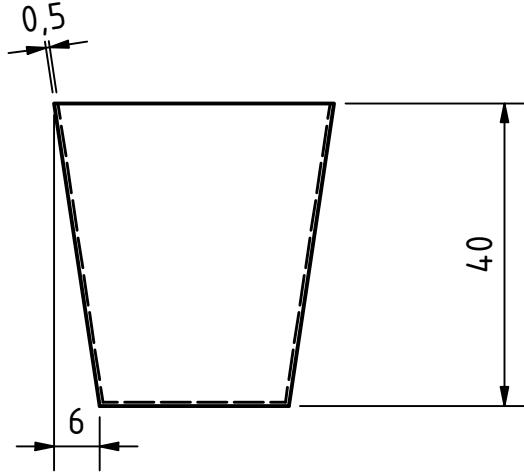
Artikelnr.:

Bauteilzeichnungsnr.: Schnappdeckelglas-1

Format: A4	Maßstab: 1 : 1	Änd.: 0	Datum: 24/10/2023	Spr.: de	Blatt: 1/1
------------	----------------	---------	-------------------	----------	------------

The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksurteileintragung vorbehalten. Schutzzertifikat nach ISO 16016 beachten.



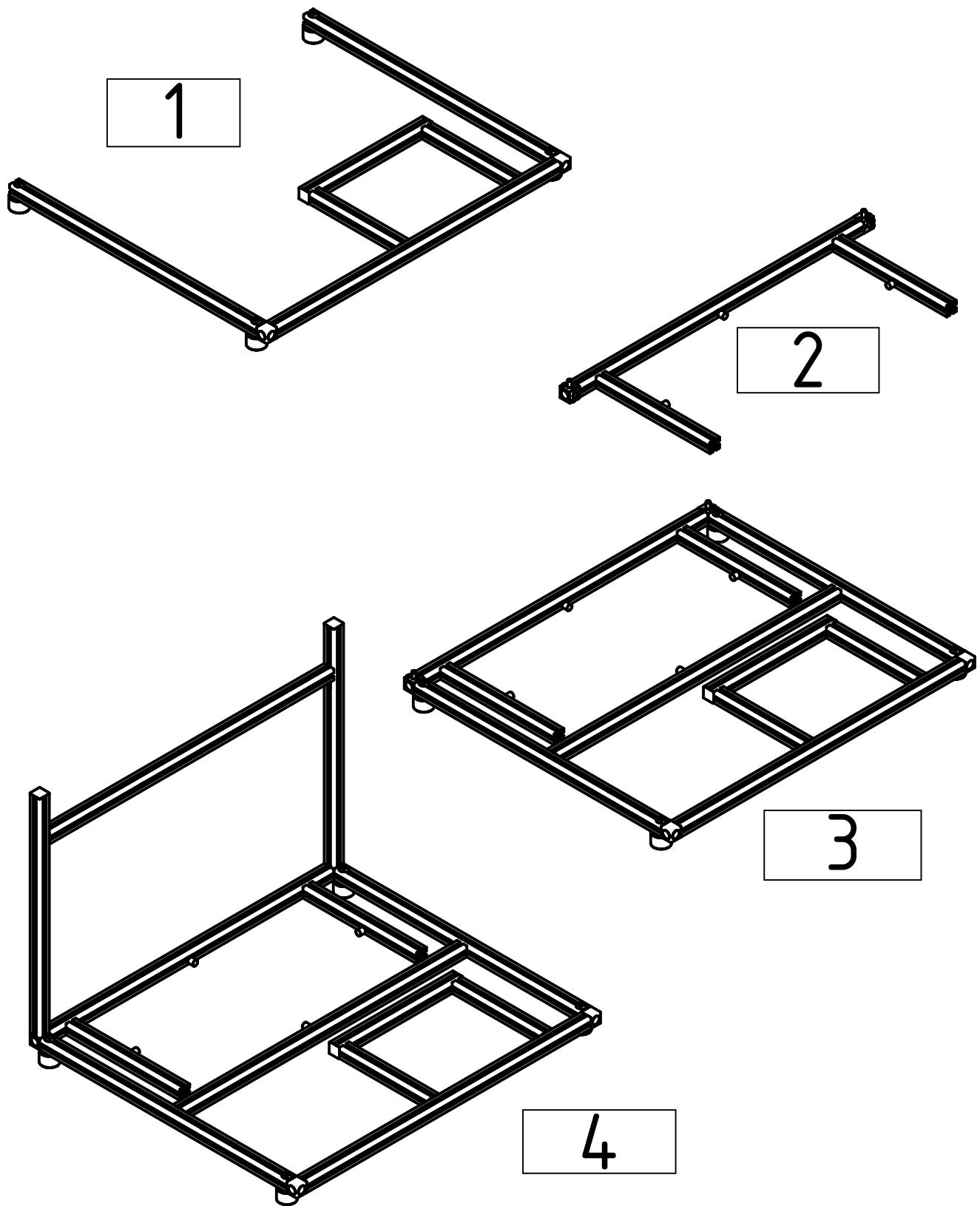
Allgemeintoleranz: DIN ISO 2768-mK	Alle Kanten R0,1 - 0,3 entgratet			
Erstellt durch: Abdelrahman Mostafa	Genehmigt von: N/A	Gewicht: 	Werkstoff: Generisch	
Bauteilbezeichnung:				
FLUXANA® XRF Application Solutions	Projekt:	Artikelnr.:		
		Bauteilzeichnungsnr.: Boromat_tiegel-1		
Format: A4	Maßstab: 1 : 1	Änd.:	Datum: 24/10/2023	Spr.:
		0		Blatt: 1/1

Appendix F

Frame Technical Drawing

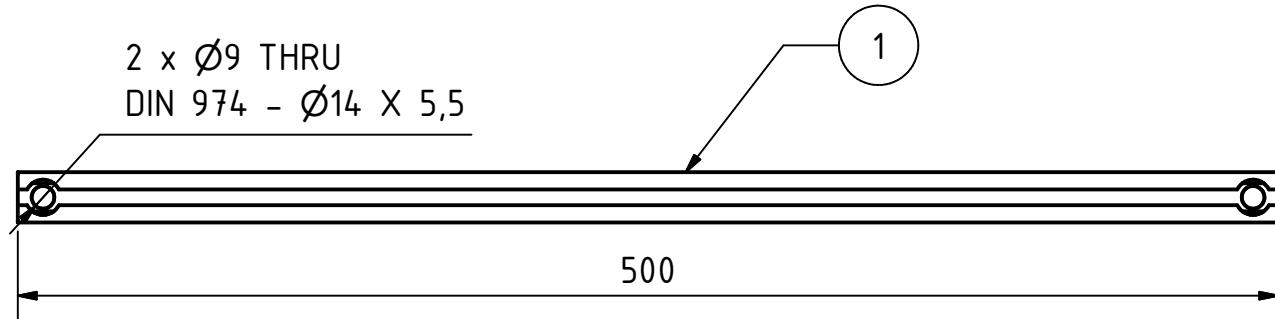
The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksmustereintragung vorbehalten. Schutzvermerk nach ISO 16016 beachten.

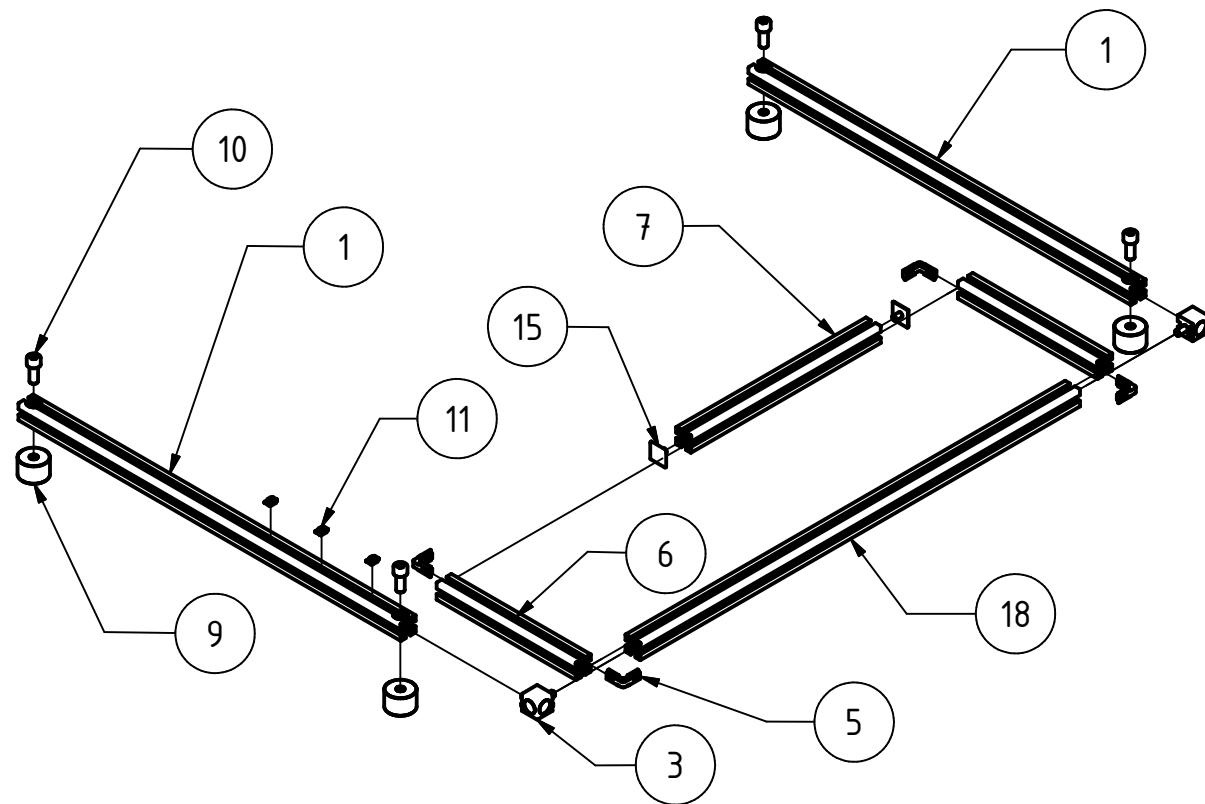


Erstellt durch: Carine Allen	Genehmigt von: 	Gewicht: N/A		
Montagebaugruppenbez.: Niryo Frame				
FLUXANA® XRF Application Solutions	Projekt:		Artikelnr.:	
Montagebaugruppenzeichnungsnr.:				
Format: A4	Maßstab: 1 : 8	Änd.:	Datum: 21/06/2023	Spr.:
		0		Blatt 1/5

Punkt 1: 2 x Bohrung im Profil 20x200x500mm



Schritt 1: Niryo holder



Erstellt durch: Genehmigt von: Gewicht:
Carine Allen N/A



Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

Projekt:

Artikelnr.:

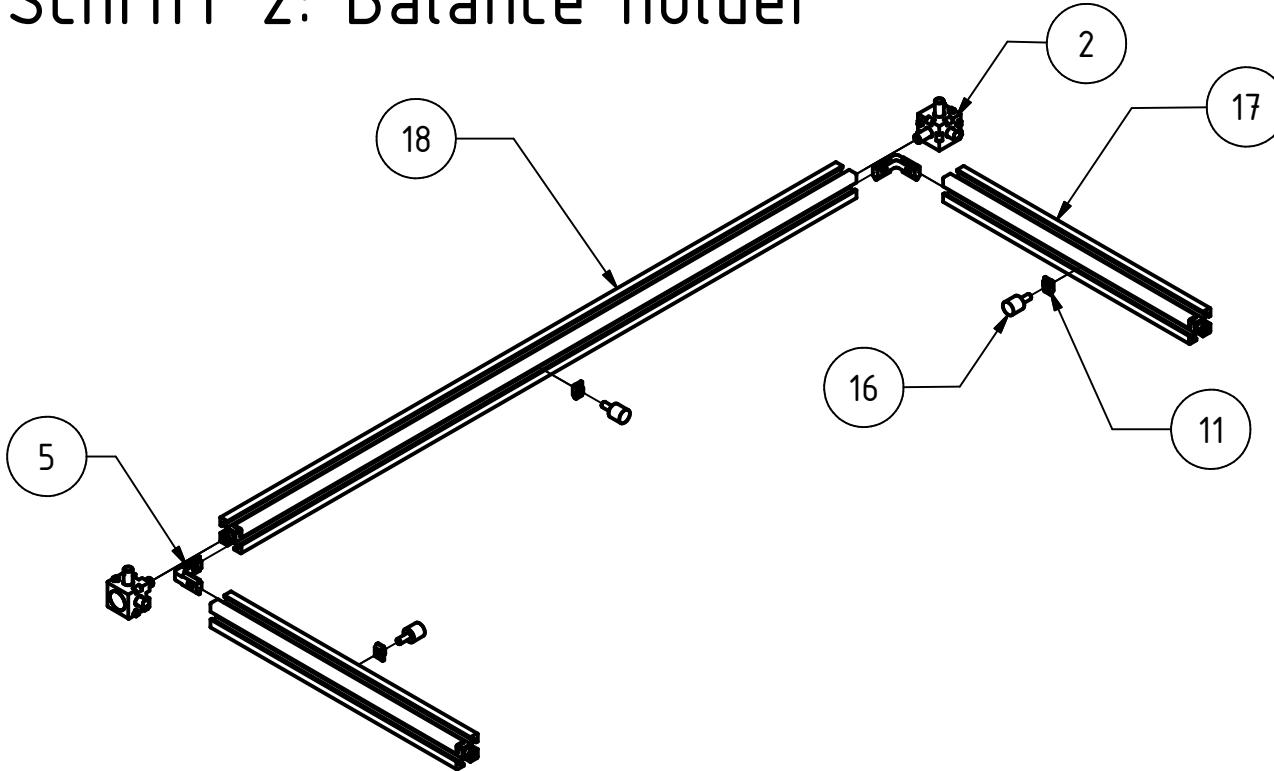
Montagebaugruppenzeichnungsnr.:

Format: A4	Maßstab: 1 : 7	Änd.: 0	Datum: 21/06/2023	Spr.: de	Blatt: 2/5
------------	----------------	---------	-------------------	----------	------------

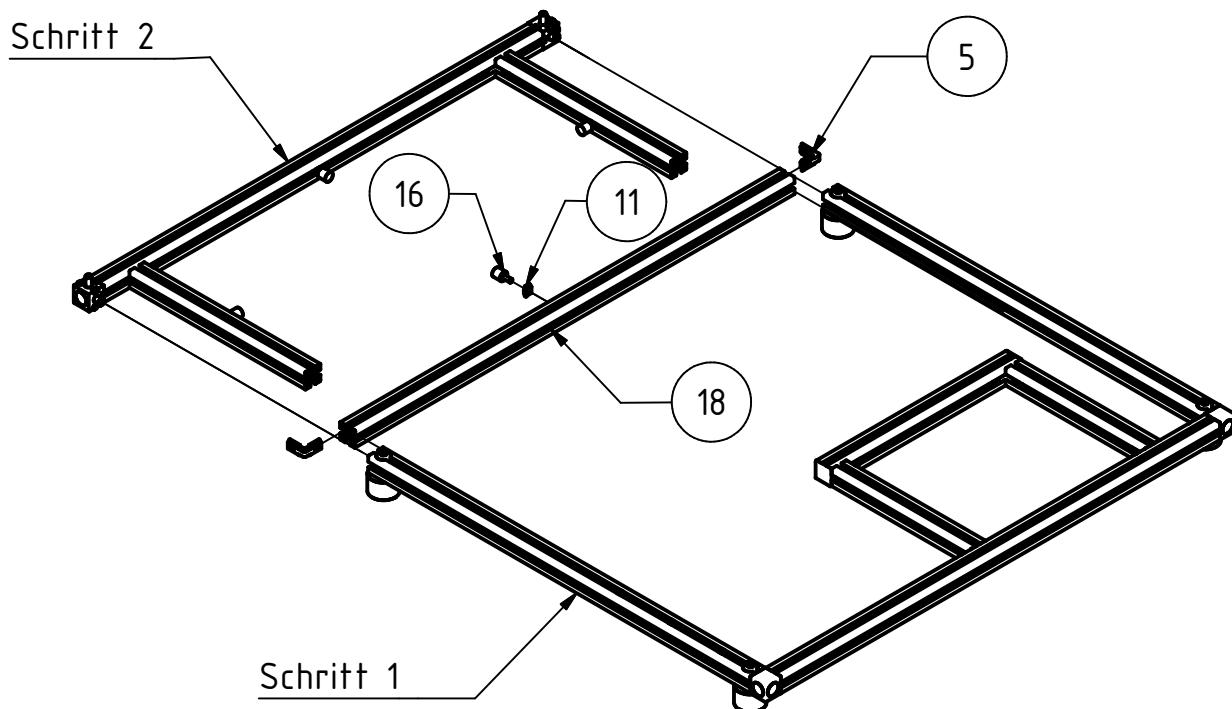
The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksmustereintragung vorbehalten. Schutzvermerk nach ISO 16016 beachten.

Schritt 2: Balance holder



Schritt 3: Base frame



Erstellt durch: Carine Allen	Genehmigt von:	Gewicht: N/A		
Montagebaugruppenbez.: Niryo Frame				
FLUXANA® XRF Application Solutions	Projekt:		Artikelnr.:	
Montagebaugruppenzeichnungsnr.:				
Format: A4	Maßstab: 1 : 5	Änd.: 0	Datum: 21/06/2023	Spr.: de Blatt 3/5

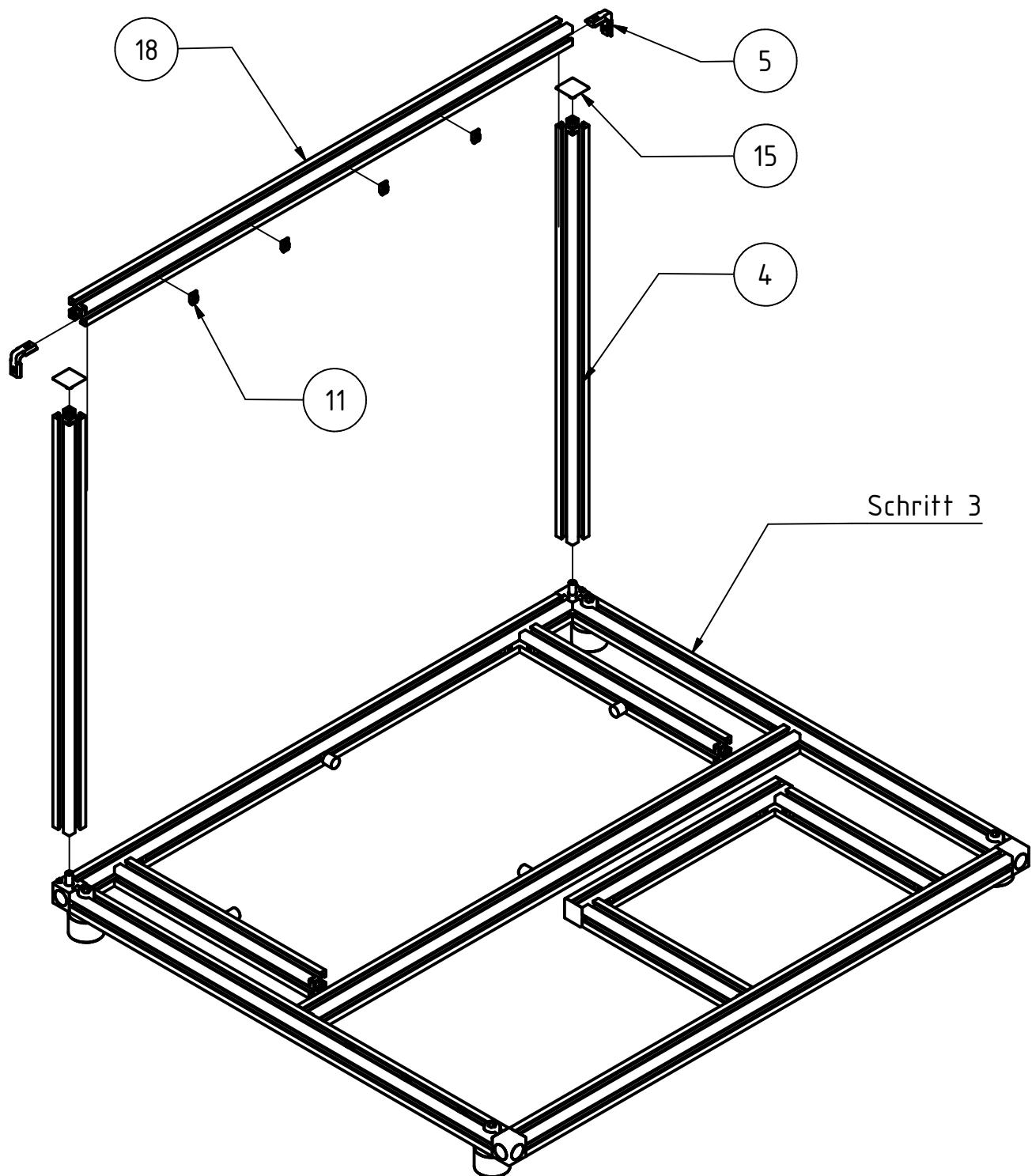
The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksmustereintragung vorbehalten. Schutzvermerk nach ISO 16016 beachten.

Schritt 4: Complete frame

The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksmustereintragung vorbehalten. Schutzvermerk nach ISO 16016 beachten.



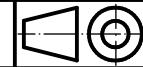
Erstellt durch: Carine Allen	Genehmigt von:	Gewicht: N/A		
Montagebaugruppenbez.: Niryo Frame				
FLUXANA® XRF Application Solutions	Projekt:	Artikelnr.:		
Montagebaugruppenzeichnungsnr.:				
Format: A4	Maßstab: 1 : 5	Änd.:	Datum: 21/06/2023	Spr.:
		0		Blatt 4/5

Montagebaugruppenstückliste

Pos.	Anz.	Artikelnr.	Norm-/ BT-nr	Beschreibung
1	2	VI-0342		Strebenprofil 20x20x500mm
2	2	VI-0348		Würfelverbinder 20/3
3	2	VI-0347		Würfelverbinder 20/2
4	2	VI-0342		Strebenprofil 20x20x420mm
5	10	VI-0344		Innenwinkel 6R
6	2	VI-0342		Strebenprofil 20x20x182mm
7	1	VI-0342		Strebenprofil 20x20x248mm
8	1		BG0106	Schublade mit Waage
9	4	BO-0222		Gummipuffer 30x20mm M8x8
10	4		ISO 4762 - M8 x 20	Hexagon Socket Head Cap Screw
11	11	KT-06-0012		Hammermutter 6 M4
12	4		Becherglass_hanger	
13	1		Schnappdeckelglas	
14	1	BO-0289	BT0953	Adapterring Schnappdeckelglas Boramat 30
15	4	KT-07-0001		Abdeckkappe grau 20x20
16	4	GR-0186		Gummipuffer D10 x H10 M4x10AES
17	2	VI-0342		Strebenprofil 20x20x230mm
18	4	VI-0342		Strebenprofil 20x20x575mm
19	3		Schnappdeckelglas_hanger	

The distribution and reproduction of this drawing, the exploitation and communication of its contents are prohibited unless expressly permitted. Any infringement shall give rise to an obligation to pay damages. All rights reserved in the event of patent, utility model or design registration. Observe protection notice according to ISO 16016.

Weitergabe sowie Vervielfältigung dieser Zeichnung, Verwertung und Mitteilung seines Inhalts sind verboten, soweit nicht ausdrücklich gestattet. Zu widerhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall der Patent-, Gebrauchsmuster- oder Geschmacksmustereintragung vorbehalten. Schutzvermerk nach ISO 16016 beachten.

Erstellt durch:	Genehmigt von:	Gewicht:	
-----------------	----------------	----------	---

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

Projekt:	Artikelnr.:				
	Montagebaugruppenzeichnungsnr.:				
	Format:	Maßstab:	Änd.:	Datum:	Spr.:
	A4	0	21/06/2023	de	Blatt 5/5

Bibliography

- [1] Robert O Ambrose et al. "Robonaut: NASA's space humanoid". In: *IEEE Intelligent Systems and Their Applications* 15.4 (2000), pp. 57–63.
- [2] Arsalan Anwar. *simple steps to create and build your first ROS package*. <https://medium.com/swlh/7-simple-steps-to-create-and-build-our-first-ros-package-7e3080d36faa>. Published in. 2021,
- [3] Stephen Brawner. *SolidWorks to URDF Exporter*. http://wiki.ros.org/sw_urdf_exporter. 2021.
- [4] Tessa Brazda. *NASA Robonaut first generation*. <https://www.nasa.gov/technology/r1-the-first-generation/>. found in. 2023.
- [5] Praneel Chand. "Developing a Matlab Controller for Niryo Ned Robot". In: *2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA)*. IEEE. 2022, pp. 1–5.
- [6] Davet Coleman. *rqt_graph*. https://wiki.ros.org/rqt_graph. (2018).
- [7] David Coleman et al. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study". In: (Apr. 2014).
- [8] Peter Corke. "Integrating ros and matlab [ros topics]". In: *IEEE Robotics & Automation Magazine* 22.2 (2015), pp. 18–20.
- [9] Anish Devasia. *PLC Sequencer Logic: An Overview*. <https://control.com/technical-articles/plc-sequencer-logic-an-overview/>. Aug. 2020.
- [10] Farbod Fahimi. *Autonomous robots*. Springer, 2009.
- [11] Carol Fairchild and Dr. Thomas Harman. *ROS Robotics By Example*. Packt Publishing, 2016. ISBN: 978-1-78217-519-3.
- [12] Tully Foote. *Topics - ROS Wiki*. <http://wiki.ros.org/Topics>. 2019.
- [13] Marc-Henri Frouin. *Ned2 Documentation*. <https://docs.niryo.com/product/ned2/v1.0.0/en/source/introduction.html>. found in. 2022.
- [14] Marc-Henri Frouin. *Niryo Company*. <https://niryo.com>. found in. 2017.
- [15] Marc-Henri Frouin. *package Documentation*. https://docs.niryo.com/dev/ros/v4.1.1/en/source/stack/high_level.html. found in. 2022.
- [16] Patrick Goebel. *ROS By Example*. Ed. by Lulu. Lulu, 2013. URL: \url{http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-indigo-volume-1/ebook/product-22015937.html}.
- [17] Damith Herath and David St-Onge. *Foundations of Robotics: A Multidisciplinary Approach with Python and ROS*. Springer, 2022.
- [18] Intel. *Types of Robots: How Robotics Technologies Are Shaping Today's World*. <https://www.intel.com/content/www/us/en/robotics/types-and-applications.html>. found in. 2023.

- [19] “International Standard ISO-17506:2022”. In: *COLLADATM digital asset schema specification for 3D visualization of industrial data*. Industrial automation systems and integration. 2022.
- [20] Reza N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. 3rd edition. Cham: Springer International Publishing, Imprint: Springer, 2022. DOI: [10.1007/978-3-030-93220-6](https://doi.org/10.1007/978-3-030-93220-6).
- [21] Lentin Joseph. *Learning Robotics using Python*. Ed. by Packt Publishing. Packt Publishing, May 2015. ISBN: 9781783287536. URL: \url{https://www.packtpub.com/application-development/learning-robotics-using-python}.
- [22] Lentin Joseph. *ROS Robotics Projects*. Packt Publishing, 2017. ISBN: 978-1-78355-471-3.
- [23] Lentin Joseph and Aleena Johny. “Getting Started with Ubuntu Linux for Robotics”. In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 1–52.
- [24] Lentin Joseph and Aleena Johny. “Programming with ROS”. In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 173–240.
- [25] Ltd. Kawasaki Heavy Industries. *Comparison between robot and human movement*. <https://robotics.kawasaki.com/ja1/xyz/en/1804-03/>. 2018.
- [26] Charles C Kemp et al. “Humanoids”. In: *experimental psychology* 56 (2009), pp. 1–3.
- [27] Jay LaCroix. *Mastering Ubuntu Server: Master the art of deploying, configuring, managing, and troubleshooting Ubuntu Server 18.04*. Packt Publishing Ltd, 2018.
- [28] B.A. Lee M. “Robotics Background Information and Facts”. In: *Salem Press Encyclopedia of Science* (2022). URL: <https://bvyg.short.gy/robotics-background>.
- [29] Kevin M Lynch and Frank C Park. *Modern robotics*. Cambridge University Press, 2017. URL: www.cambridge.org/9781107156302.
- [30] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. “On the benefits of making robotic software frameworks thin”. In: *International Conference on Intelligent Robots and Systems-Workshop for Measures and Procedures for the Evaluation of Robot Architectures and Middleware at IROS'07*. Vol. 2. 2007.
- [31] Mark Moll et al. “Teaching motion planning concepts to undergraduate students”. In: (2011). DOI: [10.1109/ARSO.2011.6301976](https://doi.org/10.1109/ARSO.2011.6301976).
- [32] Habib Oladepo. *Nodes - ROS Wiki*. <http://wiki.ros.org/Nodes>. (2018).
- [33] Janis Osis and Uldis Donins. “Chapter 1 - Unified Modeling Language: A Standard for Designing a Software”. In: *Topological UML Modeling*. Ed. by Janis Osis and Uldis Donins. Computer Science Reviews and Trends. Boston: Elsevier, 2017, pp. 3–51. ISBN: 978-0-12-805476-5. DOI: <https://doi.org/10.1016/B978-0-12-805476-5.00001-0>. URL: <https://www.sciencedirect.com/science/article/pii/B9780128054765000010>.
- [34] Jia Pan, Sachin Chitta, and Dinesh Manocha. “FCL: A general purpose library for collision and proximity queries”. In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.
- [35] Martin Pecka. *Urdf: Universal robotic description format*. <http://wiki.ros.org/urdf>. 2023.

- [36] Jan Perez Ruiz Alexander Rosell Gratacòs. "A roadmap to robot motion planning software development". In: *Computer applications in engineering education* 18.4 (2010), pp. 651–660. DOI: [10.1002/cae.20269](https://doi.org/10.1002/cae.20269).
- [37] Michael Peshkin and J Edward Colgate. "Cobots". In: *Industrial Robot: An International Journal* 26.5 (1999), pp. 335–341.
- [38] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System.* " O'Reilly Media, Inc.", 2015.
- [39] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan. 2009, p. 5.
- [40] Open Robotics. *ROS*. <https://www.ros.org/>. (2021).
- [41] Open Robotics. *ROS-documentation*. <https://docs.ros.org/>. 2015.
- [42] Isaac Saito. *Actionlib - ROS Wiki*. <http://wiki.ros.org/actionlib>. (2018).
- [43] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.
- [44] Ruben Smits, H Bruyninckx, and E Aertbeliën. *KDL: Kinematics and dynamics library*. <https://www.orocos.org/kdl>. found in. 2011.
- [45] Andrew Spielberg et al. "Functional co-optimization of articulated robots". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 5035–5042.
- [46] Adam A. Stokes et al. "A Hybrid Combining Hard and Soft Robots". In: *Soft Robotics* 1.1 (2014), pp. 70–74. DOI: [10.1089/soro.2013.0002](https://doi.org/10.1089/soro.2013.0002).
- [47] Ioan A. Sucan and Sachin Chitta. *MoveIt*. <https://moveit.ros.org/>. found in. 2013.
- [48] Niryo Team. *Control Ned with an Arduino Board*. https://docs.niryo.com/applications/ned/v1.0.3/en/source/tutorials/control_ned_arduino.html. found in. 2022.
- [49] Niryo Team. *Ned2 Overview*. <https://niryo.com/products-cobots/robot-ned-2/>. found in. 2022.
- [50] Niryo Team. *PyNiryo OpenCV Example*. https://docs.niryo.com/dev/pyniryo/v1.1.2/en/source/vision/image_processing_overview.html. found in. 2023.
- [51] Niryo Team. *Visual picking with Artificial Intelligence using TensorFlow*. <https://niryo.com/learn-robotics/visual-picking-with-ai-tensor-flow/>. found in. 2023.
- [52] Ricardo Téllez, Alberto Ezquerro, and Miguel Ángel Rodríguez. *ROS in 5 Days: Entirely Practical Robot Operating System Training*. Independently published, 2016. ISBN: 1520138733.
- [53] Ricardo Tellez, Alberto Ezquerro, and Miguel Angel Rodriguez. *ROS Manipulation in 5 Days: Entirely Practical Robot Operating System Training*. Independently published, 2017. ISBN: 1549531964.
- [54] Inc. Teradyne. *Universal Robots*. <https://www.universal-robots.com/>. found in. 2005.
- [55] METTLER TOLEDO. *METTLER TOLEDO ME BALANCE*. https://bvyg.short.gy/me_balance. 2020.

- [56] Frantisek Trebuňa. *Applied mechanics and mechatronics*. ISSN 1662-7482. Trans Tech Publications, 2014. ISBN: 3-03826-572-1.
- [57] Günter Ullrich et al. "Automated guided vehicle systems". In: *Springer-Verlag Berlin Heidelberg*. doi 10 (2015), pp. 978–3.
- [58] Lucas Walter. *roscore*. <https://wiki.ros.org/roscore>. (2019).
- [59] Yanqing Wu. *ROS-Master*. <https://wiki.ros.org/Master>. (2018).
- [60] Keenan A Wyrobek et al. "Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot". In: *2008 IEEE International Conference on Robotics and Automation*. IEEE. 2008, pp. 2165–2170.
- [61] Tatu Ylonen. "SSH-secure login connections over the Internet". In: *Proceedings of the 6th USENIX Security Symposium*. Vol. 37. 1996, pp. 40–52.